TOWARD PREDICTABLE EXECUTION OF REAL-TIME WORKLOADS ON MODERN GPUS

BY

JAYATI SINGH

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois Urbana-Champaign, 2021

Urbana, Illinois

Adviser:

Adjunct Professor Marco Caccamo

ABSTRACT

Over the last decade, real-time systems have witnessed a major increase in computational demands, which cannot be met by existing multi-core processors. Graphics processing units (GPUs) are a cost-effective solution to serve such systems. The high throughput and energy efficiency offered by GPUs has led to their widespread adoption. Most real-time systems today have multiple tasks utilizing the GPU, and GPUs are getting bigger (more processing units) with every generation. Hence, prior solutions that give each task exclusive access to the GPU are no longer feasible from a real-time as well as cost perspective. This necessitates predictable GPU multi-tasking, which unfortunately cannot be trivially achieved in modern GPUs. New spatial and temporal scheduling policies need to be explored and enforced in modern GPUs to enable predictable execution of GPU tasks. Therefore, this thesis investigates two approaches to achieve predictable execution on NVIDIA GPUs.

The first approach involves executing different tasks on disjoint sets of GPU processing units, that is, spatial partitioning (SP). There has been considerable effort by the industry and research community to enable GPU SP. However, leveraging SP to improve schedulability still needs to be investigated thoroughly. Therefore, we propose heuristics to partition the GPU into sets of processing units and assign tasks to each partition, with a goal of increased utilization while respecting the tasks' timing constraints.

The second approach to enforce multi-tasking on GPUs is simultaneous multi-kernel (SMK). SMK arbitrates between tasks at the lowest level of execution, namely, at the warp level. We propose a real-time priority aware warp scheduler and study its performance when compared against kernel agnostic policies like loose-round-robin and greedy-then-oldest, which are implemented in NVIDIA hardware today. We implement and evaluate our proposed warp scheduling policy on GPGPU-Sim.

ACKNOWLEDGMENTS

I give the most gratitude to my adviser, Prof. Marco Caccamo. I would literally not be here had it not been for his belief in my abilities. Marco has always given me precious advice, technical and otherwise, when I needed it the most. I would also like to thank Prof. Sayan Mitra for being my academic adviser.

I am grateful to Ayoosh Bansal and Nacho Sañudo for being wonderful colleagues, mentors and friends. Ninety-nine percent of all my systems and architecture knowledge comes from them. I thank them for patiently answering all of my million questions. It has truly been a privilege to work with them.

I want to thank everyone in my research group, over at TUM and here at UIUC, for all the guidance, inspiration, support and coffee.

Lastly, I thank my parents for tolerating me for the last twenty-four years and also for their unconditional love.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

API	Application Programming Interface
CKE	Concurrent Kernel Execution
COTS	Commercial-Off-The-Shelf
FIFO	First-In First-Out
FLOPS	Floating-point Operations Per Second
GPC	Graphics Processing Cluster
GPGPU	General Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
GTO	Greedy-Then-Oldest
HPC	High-Performance Computing
LLC	Last-Level Cache
MIG	Multi-Instance GPU
QoS	Quality of Service
RT-PAWS	Real-Time Priority Aware Warp Scheduling
SM	Streaming Multiprocessor
SMK	Simultaneous Multi-Kernel
SP	Spatial Partitioning
TB	Thread Block
WCET	Worst-Case Execution Time

CHAPTER 1

INTRODUCTION

Real-time systems are employed in environments where computers interact with the physical world. The defining attribute of a real-time system is that it must respond to an environmental stimulus or input within a deadline consistently. Given a real-time system comprising application *tasks* and *deadlines*, the system designer needs to (i) formulate a scheduling algorithm with assumptions on the worst-case execution time (WCET) of all tasks, (ii) analytically prove that the system deadline constraints are met with the proposed scheduling algorithm, and (ii) port the applications and the scheduling software on the selected hardware platform such that the assumptions on the tasks' WCETs hold under any circumstances, i.e. ensure *predictable execution* of the tasks on the chosen hardware.

In contrast to high-performance computing (HPC), which focuses on the processing done per unit time i.e. *throughput*, real-time computing focuses on execution-time *predictability*. Most hardware and software systems today are designed to achieve high throughput, often at the cost of predictability. Around the start of the 21st century, Dennard scaling appeared to break down and processor vendors could not operate within the same power envelope as processor operating frequencies were increased. This led to the transition towards multi-core solutions. In addition to introducing a new dimension (multiple processing units) to real-time scheduling research, commercial-off-the-shelf (COTS) multi-core platforms weakened the fundamental principle that the WCET of applications can be estimated in isolation. Subsequently, a plethora of real-time systems research has been done to propose scheduling algorithms [1] for multi-core platforms and eliminate sources of unpredictability from multi-core platforms [2, 3, 4, 5] up to the present day, and with good reason – multi-core platforms are here to stay.

However, over the last few years, there has been a major transition in the nature of the

software applications in real-time systems. Automotive applications in driver-assist and autonomous vehicle technologies have workloads that process sensor inputs for time-sensitive tasks like object tracking, lane-following and obstacle-avoidance [6]. Most of these automotive applications are safety-critical and have strict or *hard* deadlines. In contrast, augmented-reality/virtual-reality (AR/VR) applications like rendering, SLAM and eye-tracking [7, 8] have *soft* deadlines [9] or quality-of-service (QoS) requirements. What all these applications have in common is a high computational requirement that cannot be met by existing multi-core platforms. General purpose graphics processing units (GPGPUs, hereafter referred to as GPUs) offer exceptional computational throughput in the order of 19.5 trillion floating-point operations per second (19.5 TFLOPS) [10] with better energy efficiency than multi-core processors. GPUs are likely the only feasible and cost-effective solution to meet the computational demands of such applications.

GPUs released by NVIDIA have been recently deployed in many latency-sensitive systems. The NVIDIA CUDA application programming interface (API) allows the programmer to easily exploit the processing power provided by these massively parallel accelerators and is one of the major reasons behind the ubiquity of NVIDIA GPUs. Unsurprisingly, the hardware and software stack of NVIDIA is proprietary and is highly optimized for high throughput – at the high cost of reduced predictability. This forces real-time systems researchers to rely blindly on the opaque NVIDIA schedulers to arbitrate between GPU tasks called *kernels*, thereby making it difficult to estimate and/or bound the WCET of GPU-sharing tasks. With every generation, the number of processing units, called streaming multiprocessors (SMs) in GPU architectures is steadily increasing, thus exclusive access to the GPU by each task as proposed by prior work [11, 12] is no longer efficient or feasible if tasks need to meet deadlines.

There are three common approaches for multi-tasking in GPUs: (i) modify kernels to yield control of the GPU, called *cooperative multi-tasking*, (ii) assign disjoint sets of processing units (SMs) to different kernels, analogous to setting the affinity of processes in CPUs, called *spatial multi-tasking*, and (iii) arbitrate between tasks at a finer warp-level (defined in Chapter 2) granularity within each SM, similar to simultaneous multi-threading in CPUs, hence called *simultaneous multi-kernel (SMK)* by related work [13].

Since cooperative multi-tasking proposed by [14, 15] requires extensive source code modification and suffers from high context-switch latencies, our work investigates the latter two ideas with a goal of providing real-time guarantees. We propose deadline-aware spatial partitioning heuristics. Furthermore, we explore priority-based fine-grained multi-tasking on GPGPU-Sim [16] with an intention to encourage NVIDIA to enhance the CUDA Toolkit, i.e. the API, compiler, run-time libraries and driver, to provide more control to the developer, moving us one step closer to GPU predictability.

1.1 Scope and Contributions

The goal of this work is to improve the execution time predictability of kernels on NVIDIA GPUs. We present two approaches to achieve this goal.

Contention-Aware Spatial Multi-Tasking[1] We present partitioning and task-to-partition allocation heuristics that accounts for the task deadlines. We characterize kernels as memory or compute intensive and use this classification to predict the interference between co-running tasks and model the execution times of the kernels with and without interference. This information about the kernel execution times guides the allocation algorithm to find a solution that increases GPU utilization and reduces the deadline miss-rate significantly when compared to the baseline mechanism in which the GPU is a non-partitioned resource.

Priority-Based Fined-Grained Multi-Tasking We present an architectural solution that enforces kernel priorities at the lowest level of the scheduling hierarchy in GPUs. *Warps* are the smallest schedulable entity in NVIDIA GPUs. Our solution implements a real-time priority aware warp scheduler (RT-PAWS) to enforce kernel priorities during concurrent kernel execution in GPUs. Since the NVIDIA ecosystem presents no API calls to enforce this in their GPUs, we implement this on GPGPU-Sim.

While the above-mentioned principles can be extended to any computing platform, we restrict our scope to only NVIDIA GPUs in this work. NVIDIA is the leading vendor of GPUs

[1]This work was presented at The 28th International Conference on Real-Time Networks and Systems.

and has several platforms that are specifically designed for autonomous systems, for example, the NVIDIA Jetson and DRIVE PX series [17]. AMD GPUs are an alternative to NVIDIA platforms; however, the CUDA API has been adopted widely, with powerful libraries like CuDNN which serve as the backend for many prominent frameworks like TensorFlow and PyTorch. In fact, NVIDIA holds 82% of the market share at the time of writing [18].

It is worth noting that AMD provides a tool `hipify` [19] to transform CUDA code to run on AMD GPUs. AMD also provides an open-source software stack, Radeon-Open-Compute (ROCm) which simplifies WCET analysis and verification for real-time systems. AMD presents a promising alternative to NVIDIA GPUs, and if the NVIDIA ecosystem continues to remain as opaque as it presently is, it might be worthwhile for real-time system designers to explore AMD GPUs [20].

1.2 Thesis Structure

Chapter 2 reviews the necessary background in real-time systems and NVIDIA GPU architecture. Next, Chapter 3 focuses on the spatial partitioning based solution to improve predictability on NVIDIA GPUs. Chapter 4 describes an implementation of fine-grained multi-tasking on GPGPU-Sim. Chapter 5 concludes the work and outlines the plan for future work.

CHAPTER 2

BACKGROUND

We describe the programming model and the programmer-visible architecture in Section 2.1. In Section 2.2, we discuss the basic requirements for building a real-time system.

2.1 NVIDIA GPU Architecture

CUDA Program Structure NVIDIA GPUs act as accelerators to CPU hosts which offload parallel computational tasks called *kernels* to the GPU. CUDA programs are written in C/C++. C/C++ implementations serve as backbone libraries for other language implementations [21]. Any CUDA program involves *host code* as shown in Figure 2.1b that runs on the CPU and *device code*, shown in Figure 2.1a, that is offloaded asynchronously to the GPU. To offload computation to the GPU, the host i.e. the CPU does the following: (i) allocate GPU memory, (ii) copy CPU data into the allocated GPU memory, (iii) launch the kernel to perform computations on the GPU data, (iv) copy back the results into CPU memory.

GPU Compute Hierarchy Figure 2.2 provides an overview of the GPU architecture. The GPU consists of processing units called *streaming multiprocessors (SMs)* that perform the computations.Figure 2.2b shows the architecture within an SM. Each SM features different arithmetic logic units (ALUs). These ALUs are referred to as special function units, tensor cores, etc. depending on the primitive operations and data-types they support. They are also referred to as CUDA cores[1]. The smallest schedulable unit in a GPU is a *warp*. A warp is hardware-defined to be a set of 32 threads which execute in lock-step, i.e. following

[1]CUDA cores should not be seen an equivalent to CPU cores. They assume the same role as ALUs in the CPU pipelines. In fact, SMs can be considered to be the equivalent of CPU cores.

```
__global__ void transpose (float *out, float *in, int W, int H) {
    int col = blockIdx.x*blockDim.x + threadId.x;
    int row = blockIdx.y*blockDim.y + threadId.y;
    if(row < H && col < W)
        out[row*H + W] = in[row*W + H];
}
```

(a) Kernel/Device Code

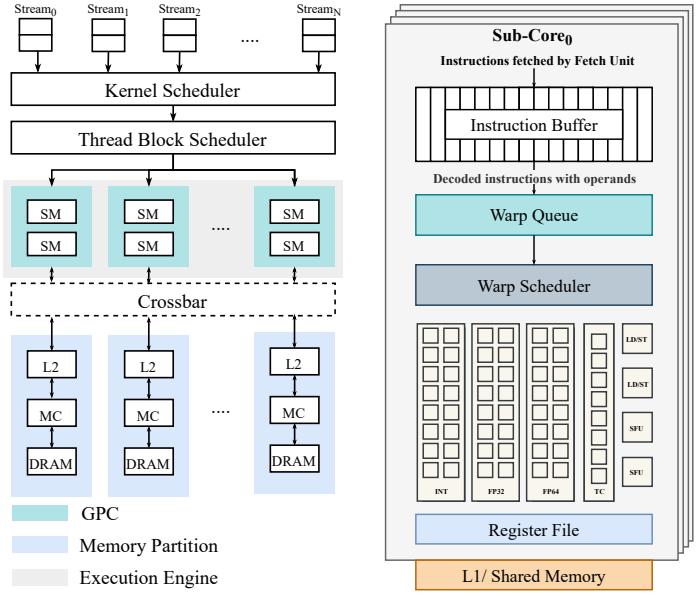
```
// Allocate and initialize data on host . . .
// Allocate pointers and memory on device
float *d_in, *d_out;
cudaMalloc(&d_in, H*W*sizeof(float)); cudaMalloc(&d_out, H*W*sizeof(float));

// Copy host initialized data over to device
cudaMemcpy(d_in, h_in, H*W*sizeof(float), cudaMemcpyHostToDevice);

// Create stream and launch kernel on stream s0
cudaStream_t s0; cudaStreamCreateWithPriority(&s0,cudaStreamNonBlocking, 0);
transpose<<<blocksPerGrid, threadsPerBlock, 0, s0>>>(d_out, d_in, W, H);

// Copy data back to host and free device memory
cudaMemcpy(h_out, d_out, H*W*sizeof(float), cudaMemcpyDeviceToHost);
cudaFree(d_in); cudaFree(d_out);
```

(b) Host Code

Figure 2.1: Code snippets for launching a naive transpose kernel on a GPU using CUDA.

single instruction, multiple threads (SIMT) execution. Each SM consists of one or more warp schedulers that arbitrate between a pool of available warps to determine which warp will issue an instruction to the ALUs each cycle. All the SMs forming the compute hierarchy are collectively abstracted as the *execution engine (EE)*. The number of warps sharing a single SM is determined by the device limits and also by the number of threads in the kernel – this is determined by the *kernel configuration*.

Kernel Configuration The kernel configuration (defined within the triple chevrons in Figure 2.1b) determines how many threads are launched. A kernel run is performed by a *grid* of *blocks* of *threads*. A grid is defined as a set of thread blocks (`blocksPerGrid` in Figure 2.1b). A thread block (TB) is defined as a set of *threads* (`threadsPerBlock` in Figure 2.1b). All the threads in a given block are executed on a single SM until completion and threads within a block cannot migrate to another SM once they are mapped to an SM.



(a) Chip level architecture of an NVIDIA GPU. Here MC refers to the memory controller.

(b) Architecture within a streaming multiprocessor (SM)

Figure 2.2: Overview of GPU architecture

Multiple thread blocks can be scheduled on the same SM if the resource requirements of the thread blocks can be met by the SM. Thread blocks in a grid can be completed out of order. The kernel finishes execution once all the blocks have completed execution.

GPU Memory Hierarchy The GPU memory hierarchy consists of the L1 cache, L2 cache, and DRAM. Each SM has a private L1 cache, shared by all the warps executing on that SM. The L2 cache is shared by all the SMs. Figure 2.2a shows the chip level architecture of a GPU. Groups of SMs are combined to form *graphics processing clusters (GPCs)* and SMs within a GPC share a common interface to the L2 memory. The L2 memory is backed by the DRAM, also called *device memory*. The GPU also consists of other memories like constant memory (a read-only scratch-pad memory) and shared memory (a scratch-pad memory private to each SM), and these can be optionally used by kernels to improve performance. The copying of data between the host and device is performed by a DMA engine shared by multiple GPCs [22]. This DMA engine is called the *copy engine (CE)* of the GPU.

Concurrent Execution with CUDA Streams The memory copy and kernel launch operations are submitted to the GPU on same or different command queues called *streams*. Commands submitted to the same stream are performed in a first-in-first-out (FIFO) order, whereas commands on different streams are performed out of order and possibly concurrently. For example, we can pipeline memory copying and kernel execution by launching the two operations on different streams. Kernels *can* be concurrently executed if launched on different streams. While launching kernels on different streams is a necessary condition for concurrent kernel execution (CKE), it is certainly not sufficient. The order of the kernel launches is completely up to the proprietary NVIDIA driver. Careful considerations in programming need to be taken to ensure the kernels are launched concurrently, outlined in [20].

2.2 Real-Time Systems

Temporal correctness is as important as functional correctness for tasks in real-time systems. Temporal correctness refers to the ability of the tasks in the real-time system to meet their deadlines consistently. The strictness of meeting the deadlines and the consequence of missing a deadline determine if a real-time system is *hard*, *soft* or *firm* [23]. For either case, we define some key aspects that determine the real-time performance of a system.

A real-time workload consists of a collection of tasks, called a *task set*. A given task τ in the task-set of a system is assumed to be periodic with a period T (although tasks can also be aperiodic) and a deadline D. Each invocation of the task is called a job. The i^{th} invocation of a job is defined as J_i. The time taken from the instance J_i is invoked to the instance it *finishes* execution is called the *response time* R_i of job J_i. A job *meets its deadline* if its response time is less than or equal to the deadline, i.e. $R_i \leq D$. A task set is *schedulable* if all the jobs meet their respective deadlines when ordered by a *scheduling policy* on a *given* hardware.

The real-time performance or *schedulability* of a particular real-time system is determined by the (i) task-set, (ii) scheduling algorithm and (iii) hardware. However, in NVIDIA GPU systems, it is often the *feasibility* of enforcing a particular policy that is challenging and is the first problem to be solved. The *mechanism* to achieve spatial partitioning has been

recently solved with the introduction of a new NVIDIA CUDA API called Multi-Instance GPU (MIG); thus, our work (in Chapter 3) focuses on the *policy* to partition the GPU given the task deadlines. In contrast, the *mechanism* to achieve fine-grained multi-tasking through priority-based warp scheduling is not explored by the research community and is certainly not supported by the CUDA API yet; therefore, we explore this with GPGPU-Sim in Chapter 4.

CHAPTER 3

CONTENTION-AWARE SPATIAL MULTI-TASKING

3.1 Introduction

Considering that the number of SMs per GPU is increasing with every generation of NVIDIA GPUs, it is no longer efficient to utilize the GPU as a single non-partitioned resource. Relying on the NVIDIA arbitration and preemption mechanisms is not suitable for real-time workloads due to high and variable response-times of tasks and priority inversion [7, 8]. Spatial partitioning of the GPU is a solution to overcome these problems and achieve predictable response times of tasks. We briefly discuss the possible *mechanisms* to achieve spatial partitioning in NVIDIA GPUs in Subsection 3.1.1.

We rely upon these solutions to enforce our spatial partitioning *policy*. Existing work (discussed in Section 3.2) proposes spatial partitioning policies that provide coarse-grained QoS, as opposed to *real-time guarantees*. Therefore, in this work we propose novel GPU partitioning heuristics based on inter-task interference and real-time constraints of the tasks. Our contributions are outlined in Subsection 3.1.2.

3.1.1 Enforcing Spatial Partitioning

It is important to note that these techniques enable different levels of GPU partitioning; however, scheduling aspects are still a responsibility of the system engineer.

Multi-Process Server NVIDIA provides the CUDA Multi-Process Service (MPS). CUDA MPS is a software feature that allows multiple processes in different contexts to execute concurrently and to reserve a percentage of the GPU computing resources to specific applications

(clients). However, this feature presents many drawbacks in terms of spatial isolation. For instance, the memory hierarchy (i.e. memory bandwidth and caches) remains shared and therefore contended among the MPS clients.

Persistent Threads and Cache Coloring Fractional GPUs [24] propose a software implementation to partition the compute and memory hierarchy of NVIDIA GPUs. The authors implement *SM Affinity* based compute partitioning through persistent threads. The key intuition is as follows. The kernel configuration is modified to include more thread blocks than originally specified. Each thread block reads an architectural register that specifies the SM it is running on. If the thread block is scheduled on an SM that does not belong to the set of SMs in the application's partition, the thread block returns without doing any work. Consequently, only thread blocks scheduled on the application's partition remain and complete execution. The memory is partitioned by implementing cache coloring on the L2 cache of the GPU. Cache coloring is a well-known solution to partition the last-level cache on CPUs [25]. Jain et al. [24] were the first to implement this on GPUs. Needless to say, this requires reverse-engineering proprietary NVIDIA L2 cache set addressing and DRAM bank addressing. While [24] presents the ideal behavior we would want from spatial partitioning and is ground-breaking research, this approach requires significant reverse-engineering and some kernel source-code modification. This would need be to implemented for every new microarchitecture. With NVIDIA having releasing a new generation of architecture every years, this approach is extremely cumbersome and not scalable.

Multi-Instance GPU Multi-Instance GPU (MIG) allows the system engineer to effectively partition the memory and compute resources of the GPU. Specifically, the partitions can be defined at the granularity of a GPC (graphics processing cluster). For example, the A100 GPU features 7 GPCs and thus it is possible to configure the GPU with 7 static partitions, each using 1 GPC and having exclusive and disjoint compute resources (SMs) and memory resources (L2 slices). More details about MIG configurations are publicly available in the relevant NVIDIA documentation[1].

[1] `https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html`

3.1.2 Summary of Contributions

Our contributions are as follows:

- We propose an approach for the classification of kernels as memory-intensive or compute-intensive, and develop a model for the kernel execution time (Section 3.3).
- Second, using the kernel execution time model, we propose heuristics to define the size of each partition and map every task to a partition such that the timing constraints of all the tasks are met.
- Finally, we evaluate the performance of the different proposed heuristics against the case where the GPU is used as a single non-partitioned resource.

3.2 Related Work

Adriaens et al. [26] were among the first to make the case that spatial partitioning results in higher performance gains than time partitioning mechanisms. They classify workloads as compute-bound, memory/interconnect-bound and problem-size bound depending on their performance sensitivity to varying number of SMs, memory and interconnect frequency respectively. They also evaluate various compile-time spatial partitioning schemes such as even partitioning, a smart-even scheme which takes the kernel dimensions into account, a rounds partitioning scheme that attempts to minimize the number of idle SMs when the kernel is near completion, and lastly, a profile-guided partitioning scheme. The partitioning schemes are evaluated using the GPGPU-Sim simulator.

The authors of [27] present a scheme to dynamically assign SM partitions to kernels. The scheme iteratively decreasing the number of SMs assigned to a kernel to reach the smallest number of SMs for the kernel without degrading the overall performance. The remaining SMs are assigned to compute-bound workloads or power-gated according to the selected mode of operation. The classification model and solution are derived and evaluated on GPGPU-Sim [16].

Similarly, [28] also focuses on the dynamic spatial partitioning of the GPU for cloud-based systems to achieve fairness and QoS targets. The authors propose a model to predict the slowdown caused by spatial multitasking to guide the dynamic partitioning scheme. In contrast to previously proposed black-box models like [29] or white-box models like [30], [28] presents a hybrid model to predict the slowdown. The paper illustrates two partitioning schemes. The first scheme, HSM-Fair, iteratively re-partitions until the normalized progress (also known as speed-up) of each running kernel is approximately equal. HSM-QoS ensures that all high priority kernels meet their normalized progress thresholds and then maximizes for the overall system throughput.

The authors of [31] propose a software based kernel scheduler, Slate. Slate co-schedules kernels with complementary resource demands to minimize interference. Pairs of kernels are profiled offline to determine the kernels are considered complementary. Slate co-schedules the complementary kernels at run-time through spatial partitioning.

In [32], Sun et al. present SMQoS, a software based mechanism to dynamically adjust SM allocation. The proposed algorithm aims to maximize throughput using information obtained by co-running tasks. In [29], the authors present a similar SM allocation engine that relies on the slowdown of the co-running applications to determine the number of SMs allocated to each kernel. The authors propose different heuristics based on fairness and QoS.

All of the above-mentioned related work aims to improve the system throughput and/or QoS with no focus on real-time performance. Moreover, most of the proposed methods in the literature focus on pair-wise allocation methods where only two kernels are considered as concurrent elements. STGM [33] proposes spatial and temporal multitasking for periodic real-time tasks. The intuition of the SM allocation algorithm in STGM is as follows. STGM first assigns the minimum number of partitions for each task and iteratively allocates more SMs for every task that is unable to meet its deadline.

STGM schedules only one task per partition, whereas our proposed heuristics can allocate the same partition to different tasks if they do not interfere with each other, thereby increasing GPU utilization.

To the best of our knowledge, we are the first to propose spatial partitioning heuristics for GPUs that consider task interference when solving the partitioning problem, thereby

optimizing for GPU utilization while also respecting all real-time constraints. Additionally, our proposed heuristics scale well beyond the concurrent execution of only two tasks.

3.3 Modeling Kernel Execution Time

Memory access patterns of CUDA kernels can drastically impact timing performance. A CUDA kernel's characteristics and functional behavior determine its compute and memory resource requirements [34]. These requirements dictate if a given kernel is classified as *compute intensive (CI)* or *memory intensive (MI)*. There can be further classification [35, 36, 37, 38], but for the purpose of this work, we categorize all workloads as either *CI* or *MI*. Intuitively, a kernel is said to be memory intensive if a significant amount of its execution time is spent in accessing memory. Conversely, it is defined as compute intensive, when most of its cycles are spent in computing instructions.

When co-running kernels compete for the same resources, their executions *interfere* with each other, hindering their overall performance. More specifically, if kernels are running concurrently, the magnitude of performance deterioration caused by mutual interference depends on the category (*MI* or *CI*) of each of the kernels. For instance, if two parallel kernels are classified as memory intensive, the overall system will experience strong contention on the memory hierarchy, causing notable performance deterioration. The decrease in performance would be significantly lower in the case the two kernels were of different types.

In order to implement contention-aware scheduling policies, it is crucial to find an accurate model to categorize kernels as compute or memory intensive. We define our classification model as follows.

3.3.1 Characterization of GPU Kernels

Classifying a kernel entails the characterization of its memory access pattern, which can be achieved by analyzing CUDA or PTX/SASS[2] code. However, this requires tedious effort

[2]PTX is the virtual ISA provided by CUDA and stays the same for every device. SASS is device-specific ISA and varies with each microarchitecture.

Table 3.1: Benchmarks used in Figure 3.1 and Figure 3.2. MI indicates memory intensive and CI indicates compute intensive.

Benchmark	Type
hs(hotspot) [39]	CI
pf(pathfinder) [40]	CI
trns(transpose) [39]	MI
vadd(vectoradd) [39]	MI

and a strong knowledge of GPU architectures and internals that can change with every generation. Therefore in this work, we use a simple yet effective approach to determine whether a kernel is compute or memory intensive.

It is known [38, 41] that load and store instructions to global memory, i.e. the last-level-cache (LLC) and main memory, present a significantly higher latency compared to compute instructions. Hence, as a rule of thumb, kernels characterized by large working data sets with few compute instructions will fall into the memory intensive category. Additionally, the performance of kernels belonging to such a category will therefore more strongly depend on the memory bandwidth allocated to them, rather than the compute resources (i.e. SMs) assigned to them. This is a well-known conclusion of the *Roofline Model* [42]; the throughput of memory intensive kernels is bottlenecked by the memory bandwidth, whereas the throughput of compute intensive kernels is limited by the instruction throughput. We propose a simpler method to classify kernels. Our proposed method does not require analysis of the PTX disassembly to the determine the arithmetic intensity, as needed by the Roofline Model [42]

Experiment To illustrate our intuition, we execute four benchmark kernels, varying the number of SMs assigned to each. The details of these benchmarks are outlined in Table 3.1. Figure 3.1 reports the execution times of the kernels normalized with respect the execution time of that same kernel running on a single SM. The experiments have been measured on an NVIDIA GeForce RTX2080 Ti GPU, featuring 68 SMs.

In this experiment we first launch a purely compute intensive dummy kernel on a CUDA stream. This dummy kernel has no memory instructions, ensuring that the kernel under evaluation suffers no interference at the memory hierarchy. The dummy kernel occupies

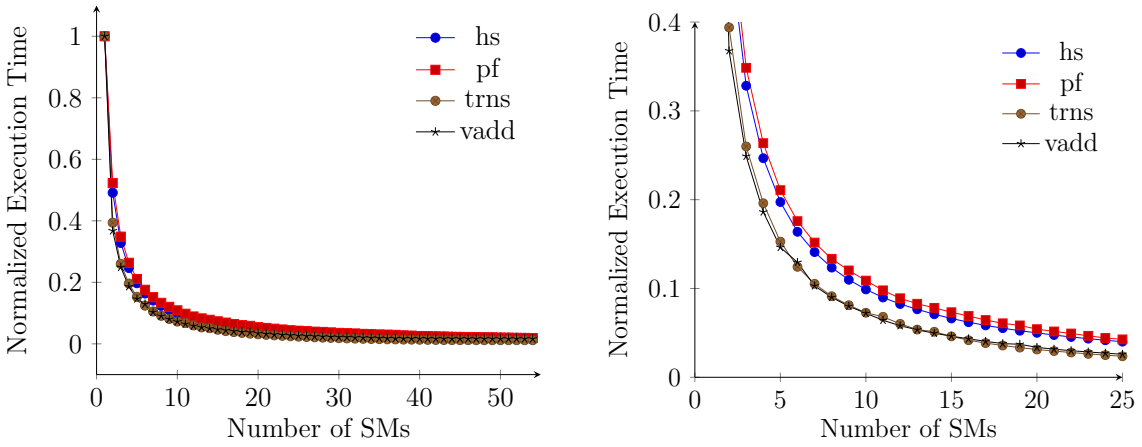


Figure 3.1: (Left) Execution time normalized with respect to a single SM when varying the number of SMs, for Hotspot, Pathfinder, Matrix Transpose and Vector Add. (Right) Normalized execution time zoomed-in.

a known and configurable number of SMs, preventing other CUDA streams (and related kernels) from occupying the same SMs as the dummy kernel. For instance, if we want to allocate m number of SMs to the kernel under test, we launch the dummy kernel so as to occupy $68 - m$ resources (there are a total of 68 SMs). We then launch the desired kernel on the unused m SMs by simply exploiting a different CUDA stream.

Results In Figure 3.1, even when the execution time of each kernel in is normalized to have the same single-SM execution time, the execution time profiles vary from kernel to kernel. We classify these kernel execution profiles into two categories. To help us identify the distinction in execution profiles more clearly, we transform the normalized execution times in Figure 3.1 into the speed-up. Therefore, we report the speed-up factors of the different kernels as a function of the number of assigned SMs in Figure 3.2

Consider the first category of kernels, which includes the **vectoradd** and the **transpose** kernels. These are identified to be memory intensive by other work [35, 36]. The second category includes **hotspot** and **pathfinder**, which are identified as compute intensive. When the number of SMs is less than 40, the increase of the number of SMs leads to linearly reduce the kernels' execution times. Starting from 40 SMs, the execution times of memory intensive kernels do not increase anymore, while those of compute intensive kernels improve linearly. In fact, for the memory intensive kernels, the GPU memory transfer bandwidth reaches

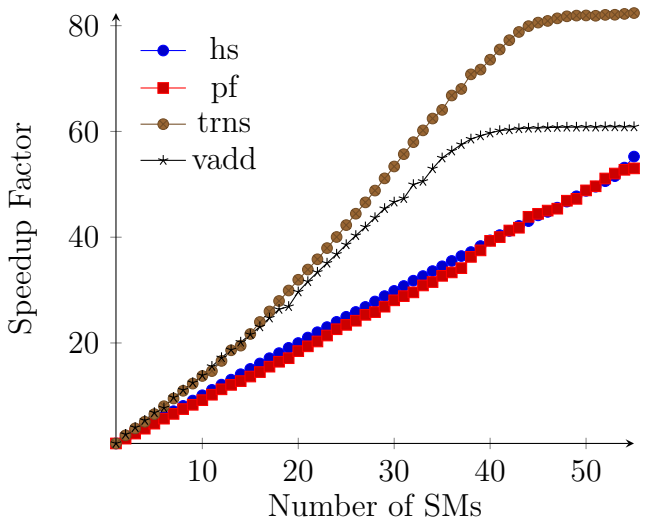


Figure 3.2: Speed-up factors of Hotspot, Pathfinder, Matrix Transpose and Vector Add as a function of the number of SMs.

saturation, using only 50% of the compute resources (SMs). As our experiments show, this does not occur in compute intensive kernels. These considerations lead to the following definition.

Definition 1. *Given a hardware architecture, a kernel is considered to be memory intensive if, beyond a certain threshold, allocating more SMs does not improve its performance on that architecture. Conversely, a kernel which is not memory intensive is considered to be compute intensive.*

We can further improve the accuracy of the definition by considering other profiling metrics such as cache hit/miss rate, however this remains as a future work.

3.3.2 WCET Model

Given real-time constraints, we aim to assign GPU partitions to concurrently running kernels. We define a *GPU partition* as a set of SMs in which we allow one or more kernels to run. Contention-aware GPU partition allocation requires us to (i) quantify the interference experienced by co-running kernels, and (ii) determine and model the effect of the interference on the kernels' response time.

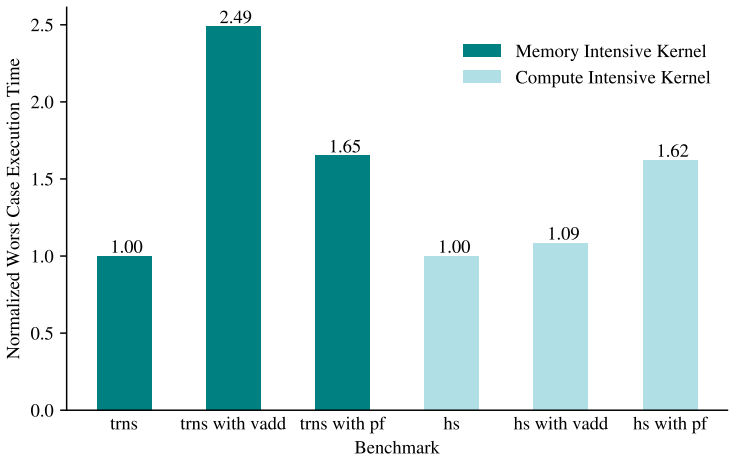


Figure 3.3: Transpose (**ts**) and hotspot (**hs**) WCET over 500 runs, with and without interference from vectoradd (**vadd**) and pathfinder (**pf**) kernels. WCETs are normalized with respect the execution times in the isolated runs without any interference.

We aim to characterize and define the magnitude of interference that a kernel experiences due to the tasks that are allocated with it on the same partition. In Figure 3.3, our experiments are set so that each kernel occupies exactly 50% of GPU resources. Specifically, this means that each kernel occupies half of *each* SM. This is possible because, depending on the kernel launch configuration (i.e., grid/blocks/threads configuration) more than one block can be scheduled concurrently within the same SM.

The first kernel we analyze is matrix transpose (**trns**) in isolation and when concurrently executed with the vector add (**vadd**) kernel. In isolation, worst case measured execution time of **trns** is equal to 2.300 ms, and the average value is equal to 2.121 ms. However, when the same kernel – which is memory intensive – runs with another memory intensive kernel (**vadd**), its execution time dramatically increases, as it goes from 2.300 ms to 5.566 ms. This decrease in performance quantifies to more than twice its execution time in isolation. However, the decrease in the **trns** kernel's performance is less dramatic when run alongside the compute intensive pathfinder (**pf**) kernel.

Next, we run the hotspot (**hs**) compute kernel, with and without the interference coming from other memory and compute intensive kernels, **vadd** and **pf**, respectively. We notice that the **hs** kernel's performance is impacted more when run along-side a compute intensive kernel, i.e. **pf**. Moreover, the slowdown is negligible in the **hs** kernel when run with the

memory intensive **vadd** kernel.

Indeed, it has been shown in [34] that compute and memory intensive kernels interleave with negligible effects on their execution times when they are executed concurrently. Therefore, it is important to define a proper strategy to allocate tasks in the same or different partitions, to reduce or eliminate the interference. For example, allocating kernels of different types to the same partition(s), thus reducing contention on similar resources, can help to increase the platform's utilization and predictability. The measurements presented above give an idea of the timing behavior of kernels running on a GPU and also confirm that our kernel classification is consistent with observed interference effects.

In this work, we present heuristics for partitioning GPU compute resources, and evaluate them on a synthetic dataset. To produce this dataset, we adopt a simple model to describe kernel behavior. This model aims not to capture all the phenomena at play during kernel execution, but simply to produce inputs similar to those observed in the measurements – which is sufficient to evaluate our heuristics satisfactorily.

Our first observation is that the kernel execution time scales as a semi-linear function of the total number of SMs. Our second observation is that for a constant number of allocated resources, the execution time of a kernel is more impacted when it is in the presence of a kernel that uses mostly the same type of resource. Rather than explicitly modeling the interference, we use two different functions to model each kernel's execution time. One function model's the execution time with little to no interference and another models the execution time with significant interference.

$$\mathcal{C}^n(m) = \frac{a^n}{m} + b^n$$

$$\mathcal{C}^c(m) = \frac{a^c}{m} + b^c$$

Here, $\mathcal{C}(m)$ is the execution time of a given kernel when executed on m number of SMs, a^n and b^n (resp. a^c and b^c) are constants describing the execution of the kernel with no or little interference or *conflict* (resp. with significant conflict). These functions will be used in Section 3.6 to evaluate the performance of our heuristics.

3.4 System Model

We discuss the GPU architecture model and the task model we use for our partitioning heuristics.

3.4.1 Architecture Model

A GPU is composed of M streaming multiprocessors (SMs). The j^{th} SM is denoted by p_j. A partition is a set of SMs. The k^{th} partition defined as $\mathcal{P}_k = \{p_{k,1}, p_{k,2}, \cdots, p_{k,|\mathcal{P}_k|}\}$ is composed of $|\mathcal{P}_k|$ SMs. Partitions do not share compute resources, i.e. $\mathcal{P}_k \cap \mathcal{P}_{k'} = \varnothing, \forall k, k \neq k'$.

We make the assumption that the considered architecture does not lead to timing anomalies. In particular, we hypothesize that the response time of a job cannot decrease when more SMs are assigned to it. This assumption is consistent with the measurements illustrated by Figure 3.2 which shows that once memory becomes the bottleneck, performance stagnates but does not decrease.

3.4.2 Task Model

Let $\mathcal{T} = \{\tau_1, \tau_2, \cdots, \tau_n\}$ be a set of kernels, henceforth referred to as *tasks*. Each task τ_i is a sequence of jobs (possibly infinite), characterized by a tuple $\tau_i = \{T_i, D_i, M_i, \mathcal{C}_i^c(m), \mathcal{C}_i^{\text{n}}(m)\}$, where

- T_i is the task period. It represents the minimum time between two successive jobs of the task τ_i.
- D_i is the relative deadline; task τ_i must complete no later than D_i time units from its arrival.
- $C_i^c(m)$ represents the task execution time when the task has at least one conflict and is running on all streaming multiprocessors in a partition of size m. A memory intensive (resp. compute intensive) task is said to be in *conflict*, if at least another memory intensive (resp. compute intensive) kernel is allocated within the same partition.

- $\mathcal{C}_i^n(m)$ represents the task execution time when it is executed in isolation, i.e. *not* in conflict, on a partition of size m.

- $\mathcal{M}_i$ denotes the task type. If the task is memory intensive, $\mathcal{M}_i$ is set to *memory*, otherwise it is set to *compute*.

We define $\mathcal{T}(\mathcal{P})$ as the set of tasks that are allocated to partition $\mathcal{P}$. We denote the execution time of task τ_i when allocated with the set of tasks $\mathcal{T}(\mathcal{P})$ on a partition $\mathcal{P}$, by $\mathcal{C}_i(\mathcal{T}(\mathcal{P}), |\mathcal{P}|)$. Thus,

$$\mathcal{C}_i(\mathcal{T}(\mathcal{P}), |\mathcal{P}|) = \begin{cases} \mathcal{C}_i^c(|\mathcal{P}|), \text{ if } \tau_i \text{ has } conflict \\ \mathcal{C}_i^n(|\mathcal{P}|), \text{ otherwise} \end{cases}$$

τ can have $conflict$, according to its type, and the number of tasks allocated with it on the same partition, as well as their type.

For example, let us a consider a GPU partition in which we assign 10 kernels, 9 of which are compute intensive. The one remaining memory intensive kernel will not experience any contention and its conflict *tag* is set for non-conflict, whereas the tags of the nine compute kernels will be set to *conflict*.

3.5 Heuristics

In the considered architecture, there are M compute resources (SMs), and our goal is to allocate tasks (kernels) to compute resources such that the overall interference is reduced, and all deadlines are met. The addressed problem is quite challenging; it takes a set of tasks as the input and is required to (i) partition the M resources to a set of partitions, selecting the suitable size for each partition and (ii) allocate tasks to partitions, such that no deadline is missed. The space of solutions is highly combinatorial, as both the SM-to-Partition and the Task-to-Partition allocations are variable.

A simpler problem of considering only the Task-to-Partition allocation has been shown to be intractable and NP-hard for large numbers of resources and tasks. In this chapter, we tackle a more complex problem as it includes not only the task to partition allocation,

but also defining the partition size. Therefore, we investigate tractable heuristics aiming to explore the solution space with a reasonable complexity (pseudo-polynomial complexity of $\theta(M \cdot N \cdot H)$ where M is the number of SMs, N is the total number of tasks and H is the task-set hyperperiod. Note that we may omit the task index when it is not necessary.

Before starting our heuristics, we apply a fast *necessary* schedulability test described in Lemma 1.

Lemma 1. *Let $\mathcal{T}$ be a task set. $\mathcal{T}$ is not schedulable if*

$$\sum_{i}^{|\mathcal{T}|} \frac{\mathcal{C}_i^n(1)}{T_i} > M$$

Proof. When the number of partitions is 1, all the M SMs are in that one partition. The proof is a straightforward derivation from classical schedulability tests based on utilization for multicore architectures.

The proposed heuristics are greedy but keep track of the solutions that have failed. We start by applying Lemma 1 to check if the task is not schedulable (not explicitly mentioned on the algorithm). The pseudo-code of the main procedure is shown in Algorithm 1. The algorithm starts by building a baseline set of partitions ($pars$), where each contains only one task, (Line 4). This list is maintained in a given order (decreasing/increasing utilization order). At each step, the algorithm selects two partitions from the partition list (Line 10) and tries to merge them in order to reduce the number of required resources (Line 17). If the merge is not possible, the algorithm stores a trace of the impossible allocation (Line 19), and tries another pair of partitions. The algorithm exits on **Success** if the total number of compute resources in $pars$ is not greater than the number of available resources, or it exits on **Fail** if it cannot select merge candidates that would allow to decrease the number of required resources.

3.5.1 Initialize Partitions

Our heuristic starts by building the baseline set of partitions $pars$. Specifically, it creates a partition for every task. As the task executes without contending with other tasks, the task

Algorithm 1 Partition and Allocate Tasks to Partitions

```
1  input 𝒯                                                          ▷ Taskset
2  input M                                          ▷ Number of compute resources (SMs)
3  function PartitionAndAllocate(𝒯, M)
4      pars ← InitPartitions                          ▷ pars: List of partitions
5      if ACT then
6          PreFill(forbiddenList)                   ▷ All pairs tested for merge offline
7      end if
8      Π ← Σ_{j=1}^{|pars|} |𝒫_j|          ▷ Total compute resources utilized by all partitions
9      while (Π > M) do
10         (𝒫^1, pars_ELIG) ← SelectPartitions(pars)   ▷ pars_ELIG: Eligible partitions
11         if 𝒫^1 = ∅ then
12             return Fail
13         end if
14         repeat
15             𝒫^2 ← Choose(pars_ELIG)
16             pars_ELIG ← pars_ELIG \ {𝒫^2}
17             𝒫^3 ← Merge(𝒫^1, 𝒫^2)
18             if 𝒫^3 = ∅ then
19                 AddToForbiddenPairs(𝒫^1, 𝒫^2)
20             else
21                 pars ← pars \ {𝒫^1, 𝒫^2}
22                 pars ← pars ∪ {𝒫^3}          ▷ every 𝒫 ∈ pars is always schedulable
23             end if
24         until (𝒫^3 ≠ ∅) ∨ (pars_ELIG = ∅)
25         Π ← Σ_{j=1}^{|pars|} |𝒫_j|
26     end while
27     return Success
28 end function
```

execution time $\mathcal{C}_i^n(\mathcal{P})$ is considered.

Lemma 2. *Let $\mathcal{P}$ be partition, such that $\mathcal{T}_\mathcal{P} = \{\tau\}$. The task τ is schedulable under $|P|$ resources if*

$$|\mathcal{P}| = \min\{m \in \{1, \cdots, M\}, \textit{such that } C^n(m) - D \leq 0, \}$$

Proof. The execution time is a decreasing function of the number of resources. Therefore, to ensure the task schedulability is sufficient to set its partition size to the smallest number of compute resources, allowing the task worst case execution time to be equal or less than the task deadline. Therefore, the partition size is set to the first number of SMs that verifies the latter condition. The goal of this step is to create for every task a partition. The condition

is that the partition has the minimum number of SMs, such that the task is schedulable. Lemma 2 is applied on every task, and every produced partition is stored in $pars$.

At every step of our heuristic, we ensure that every partition within $pars$ is schedulable. We denote by Π the number of compute resources that are required in $pars$. It is computed as follows, also shown in Lines 8 and 25 of Algorithm 1:

$$\Pi = \sum_{j=1}^{|pars|} |\mathcal{P}_j|$$

Lemma 3. *Let $\mathcal{T}$ be the task set partitioned using Algorithm 1, then $\mathcal{T}$ is schedulable if*

$$\Pi \leq M$$

Proof. The proof is very simple. Every partition is schedulable if we do not consider the other partitions. Therefore, the whole system is schedulable if the total number of required resources is less or equal to the available resources.

Lemma 3 allows to exit Algorithm 1 on success at any time of its execution. If the condition in Lemma 3 is not satisfied, we move to the next step, namely *partition merging*.

3.5.2 Partition Merging and Minimizing Resources

The goal of the second step of our heuristic is to reduce the number of required SMs for the input task set to be schedulable. Even if the number of required SMs per partition is minimal, the schedulability test in Lemma 3 is pessimistic.

Definition 2. *Let $\mathcal{P}^1(\mathcal{T}^1, m^1)$ and $\mathcal{P}^2(\mathcal{T}^2, m^2)$, be two partitions.*
$\mathcal{P}^3(\mathcal{T}^3, m^3)$ is a valid merge of $\mathcal{P}^1$ and $\mathcal{P}^2$ if,

$$\mathcal{T}^3 = \mathcal{T}^1 \cup \mathcal{T}^2, \text{ and}$$
$$m^3 < m^1 + m^2$$

The merge is denoted as $\oplus$. Therefore, $\mathcal{P}^3 = \mathcal{P}^1 \oplus \mathcal{P}^2$.

According to Definition 2, a merge is considered as valid only if it allows to allocate all the tasks of two partitions to a single new partition and the required resources to schedule the merged task set are less than the sum of the required resources for the two partitions.

Algorithm 2 Merge Partitions

```
1  input P^1, P^2                                   ▷ P^1 and P^2 are partitions.
2  function MERGE(P^1, P^2)
3      schedulable ← False
4      T^tmp ← T(P^1) ∪ T(P^2)
5      m ← max{|P^1|, |P^2|}
6      while not schedulable and m < |P1| + |P2| do
7          schedulable ← TESTSCHEDULABILITY(T^tmp, m)
8          if schedulable then
9              return P(T^tmp, m)                   ▷ Return merged partition
10         else
11             m ← m + 1
12         end if
13     end while
14     return ∅
15 end function
```

Algorithm 2 decides whether a merge is valid or not. It takes as input two partitions and produces a valid partition or fails. It first merges the two task sets (Line 4). In the first iteration, it considers a number of resources equal to the maximum number of resources between the two input partitions (Line 5). Next, it iteratively tests the schedulability of the merged task sets. If the schedulability fails, it increases the number of required resources and retests the schedulability (Line 11). This algorithm can be improved by using a binary search, applied between $\max\{|\mathcal{P}|^1, |\mathcal{P}|^2\}$ and $|\mathcal{P}^1| + |\mathcal{P}^2|$. The algorithm exits when the number of required resources m is greater than or equal to the required resources of the two partitions when considered independently.

A merge can fail because of the increase of the execution time when two conflicting tasks are merged and therefore require more resources than when the partitions are independent. It can also fail if it simply does not reduce the number of required resources (the merge leads to a number of resource that is equal to sum of resources of the two partitions); in this situation it is better to keep each partition independent, as it contains fewer tasks, and therefore allows finer merges in future iterations.

Definition 3. *Let $\mathcal{P}, \mathcal{P}'$ and $\mathcal{P}''$ be three partitions. We define partial relation order $\gg$ as follows:*

$$\mathcal{P} \oplus \mathcal{P}' \gg \mathcal{P} \oplus \mathcal{P}'' \implies |\mathcal{P} \oplus \mathcal{P}'| < |\mathcal{P} \oplus \mathcal{P}''|$$

The order relation $\gg$ allows to order two merges which have one partition in common as a function of the compute resources they use.

Definition 4. *Let $\mathcal{P}, \mathcal{P}'$ and $\mathcal{P}''$ be three partitions. We define partial relation order $>$ as follows:*

$$\mathcal{P} \oplus \mathcal{P}' > \mathcal{P} \oplus \mathcal{P}'' \implies U(|\mathcal{P} \oplus \mathcal{P}'|) < U(|\mathcal{P} \oplus \mathcal{P}''|)$$

where U is the partition utilization, defined as

$$U(\mathcal{P}) = \sum_{i=1}^{|\mathcal{P}|} C_i(\mathcal{T}^p - \{\tau_i\}, |\mathcal{P}|)$$

The order relation $>$ allows to order two merges which have one partition in common as a function of the workload that they generate. The performances of order relation $>$ and $\gg$ will be evaluated in Section 3.6.

3.5.3 Forbidden List

The forbidden list, represented by $forbiddenList$, is a list of task pairs. The tasks within the same pair *usually* require more resources to meet deadlines when allocated to the same cluster, than when allocated to different clusters. Therefore, when a merge fails, the input partitions are added to the forbidden list. The forbidden list is checked before selecting the partitions to merge within Algorithm 3.

$forbiddenList$ is first initialized as empty and is filled during analysis when a merge fails. An additional step, denoted by **ACT** in Section 3.6, fills $forbiddenList$ explicitly as a preliminary offline step. If the ACT step is skipped (denoted as **INA** in Section 3.6), every pair of tasks is tested to check if they are 'mergeable'. If the test fails, the pair is added to the $forbiddenList$.

3.5.4 Select Partitions

The partition selection process is very important to the performance of Algorithm 1 since our heuristic does not back-track the design space exploration. The partition selection (Algorithm 3) goes through three steps: (i) select the first partition $\mathcal{P}$ of the merge operation (Line 7), (ii) select all partitions eligible to merge with $\mathcal{P}$ (Line 11), and (iii) sort the list of eligible partitions according to either $\gg$ or $>$ (Line 12).

Algorithm 3 Select Partitions

```
 1 input pars                                              ▷ List of partitions
 2 input order                                             ▷ Can be ≫ or >
 3 input forbiddenList                                     ▷ List of forbidden pairs
 4 function SelectPartitions(pars)
 5     candidates ← pars
 6     repeat
 7         P ← ChooseFrom(candidates)
 8         candidates ← candidates \ {P}
 9         forbidden ← {P' | (P, P') ∈ forbiddenList ∨ (P', P) ∈ forbiddenList}
10         pars_ELIG ← pars \ forbidden
11         if pars_ELIG ≠ ∅ then
12             return (P, Sort(pars_ELIG, order)
13         end if
14     until pars = ∅
15     return ∅
16 end function
```

By definition, $\gg$ relation orders merged partitions and therefore returns only one element containing the best merge, whereas $>$ relation does not require a merge to sort partitions and therefore returns a sorted eligible partition list.

3.5.5 Schedulability Test

The scheduling policy within each partition is independent of the proposed partition and task to partition allocation. Any policy for which there is an effective schedulability test can be used. Choosing a specific scheduling policy allows us to further reduce the complexity of defining partition sizes. It also allows us to increase every partition utilization, as well as refine the selection of $pars_{ELIG}$ by applying time sensitive analysis found in [43, 44].

Independent of the scheduling policy, we highlight two approaches for taming the interference effects of conflicting kernels within the same partition. The first involves explicitly taking the interference into account for the analysis. This is far from trivial and is still a work in progress. The second, more pessimistic but simpler approach accounts for the effect of interference in the WCET estimations. This is the approach we followed in this work. Let us recall that in the considered model, the function that gives the worst case execution time of a task takes into account the absence (resp. presence) of conflict for this task within its partition to choose a duration that does not integrate (resp. integrate) the effect of interference.

3.6 Evaluation

In this section, we evaluate the schedulability of different versions of the proposed heuristics against using the GPU as a single non-partitioned resource. Furthermore, we evaluate the analysis time of our heuristics. We apply our proposed heuristics on a large number of randomly generated synthetic task-sets. The task-set generation is described in Subsection 3.4.2 and the simulation results are presented in Subsection 3.6.2. We simulate the NVIDIA GeFORCE 2080 Ti GPU as it was the primary platform for our hardware experiments. This GPU features 68 compute resource (SMs).

3.6.1 Task-Set Generation

We consider two types of task-sets:

Task-sets with 50 tasks This implies an average workload of 0.73 tasks per SM. The goal here is to select a number of tasks less than the number of SMs, so that it will force our heuristics to misbehave; i.e., the grouping capabilities of our approach may be reduced, as it will likely create a partition for every task.

Task-sets with 200 tasks This implies an average workload of 2.94 tasks per SM. Practical systems may or may not have task-sets where the number of tasks is greater than the number of SMs by this magnitude. However, such a situation is possible when using GPUs in embedded platforms where the number of SMs is small. For example, NVIDIA Jetson TX2 has only 2 SMs. In fact, it is our goal to make the execution of bigger task-sets feasible on embedded systems with the proposed heuristics. Therefore, with this experiment, we test how well our heuristics scale with a bigger task-set.
Each task-set scenario is generated as follows:

1. Generate tasks' utilization using the UUniFast-Discard algorithm [45].

2. The baseline period of every task is selected from a predefined list of periods, where the lowest period is 50, and the highest period is 4000. When selecting the task period, we make sure that a reasonable execution time can be assigned to the task. If such execution time is not reasonable, then the period is increased within the limit of 4000. The task deadline is set to $0.75\mathcal{T}_i$.

3. We obtain the baseline execution time of each task by multiplying the task period and the task utilization. We then use the execution time scaling function, defined as follows, to determine the task execution time.

The scaled execution time $\mathcal{C}_i(\mathcal{T}(\mathcal{P}), |\mathcal{P}|)$ is based on

$$\mathcal{C}_i(\mathcal{T}(\mathcal{P}), |\mathcal{P}|) = k_i \left(\frac{a_i}{|\mathcal{P}|} + b_i \right)$$

where

- a_i is the baseline execution time of τ_i.

- b_i represents the part of the execution time that does not profit from parallelism, as defined in Amdahl's law [46]. b_i is set to $0.02 \times a_i$ (resp. $0.1 \times a_i$) if τ_i is compute intensive (resp. memory intensive).

- k_i is a factor used to inflate execution times in case of conflict. Thus, it is set to 1 when τ_i has no conflict in $\mathcal{P}$, and to 1.2 (resp. 2.3) when τ_i has conflict in $\mathcal{P}$ and is compute intensive (resp. memory intensive).

We obtain the values of b_i and k_i through linear regression on the benchmarks used in Section 3.3. Furthermore, a task is chosen to be compute or memory intensive kernel according to the prm variable. A random number α is generated between 0 and1. If α is greater than prm, the task is a compute intensive kernel; otherwise it is a memory intensive kernel. prm is set to 50% for all task generation.

3.6.2 Simulation

We vary the total utilization from 2 to 68, with a step size of 2. Each point in the figures represents the average value among 100 simulations. We evaluate the schedulability rate and analysis time of the various proposed heuristics. The baseline where the GPU is considered as a single non-partitioned resource is denoted by $1G$ in Figures 3.4 and 3.5. Our heuristics can either use $\gg$ heuristics, which is denoted by SMS in the figures, or use $>$ which represents the best fit algorithm, denoted by BF. The preliminary step where all possible pairs are tested offline for a merge, to fill an exhaustive forbidden pair list, is denoted by ACT (i.e. ACT is **True** when this step is *activated*.). In contrast INA denotes the heuristics that skip this step (Line 5, Algorithm 1).

Schedulability Rate In Figure 3.4, we report the average schedulability as a function of total utilization. In this experiment, we have synthesized 100 task-sets per utilization factor, each having 50 tasks, where each task has 50% probability of being compute or memory intensive. When the total utilization is less than 35, the GPU does not reach a critical load situation, and therefore all heuristics are able to achieve 100% schedulability. Such a schedulability ratio is also reached by the baseline approach ($1G$), wherein the whole GPU is considered as one single partition. Starting from 35, the schedulability of $1G$ drastically falls, as it executes workloads close to maximal schedulable utilization. Indeed, when all tasks are in a single partition, they are probably in *conflict*; hence, their execution time with

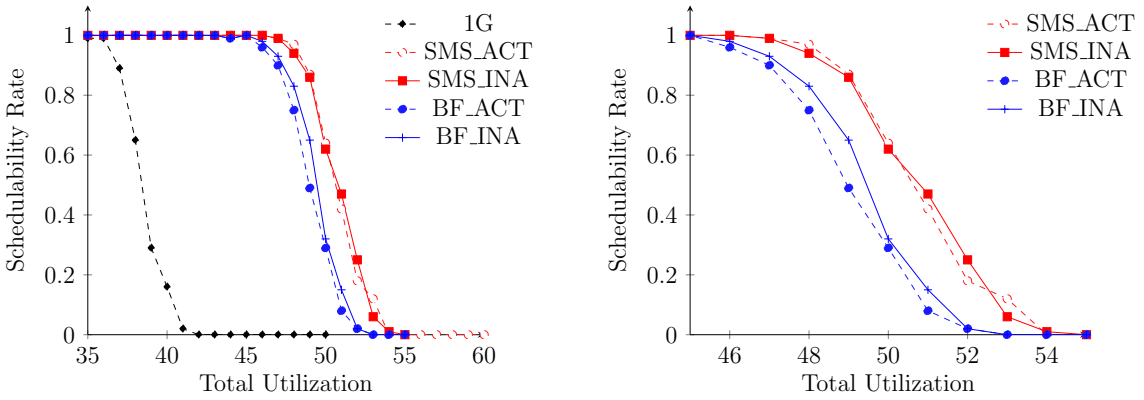


Figure 3.4: (Left) Schedulability rate for 50 tasks, with $prm = 50\%$.
(Right) Zoomed-in schedulability rates for proposed heuristics.

maximum interference is considered. We see that as the utilization increases, the $\gg$-based heuristics (SMS) perform better than the $>$-based heuristics (BF). When the exhaustive forbidden pair list is activated (ACT), the different heuristics performance slightly degrades. Indeed, the selection of candidates limits number of merge choices at early iterations of the algorithm. When it is disabled (INA), the heuristics are allowed to merge tasks that are not 'mergeable' individually. This is possible when the individual tasks belong to 'mergeable' larger partitions. This is implicitly prohibited when exhaustive forbidden pair list is used and may lead to better solutions as it explores a larger design space.

In Figure 3.5, we report the average schedulability as a function of total utilization by generating 200 tasks. Our heuristics dominate the non-partitioned GPU approach. We also note that our heuristics show performances closer to each other in Figure 3.5 than in Figure 3.4. As tasks are smaller and more numerous, our heuristics are forced to allocate more than two tasks to each partition, which leads to increasing interference and thus downgrading tasks' performance compared to the first experiment with 50 tasks per set.

In Figure 3.6, we report the analysis time as a function of total utilization. It is notable that the analysis time does not depend on the total utilization. As expected, activating the exhaustive forbidden pair list (ACT) reduces the analysis time significantly because it explores a smaller design space than INA. We also notice that the heuristics based on BF are faster than those based on SMS. This is because the $\gg$ relation used in SMS requires computing candidate merging, while the relation $>$ used in BF does not.

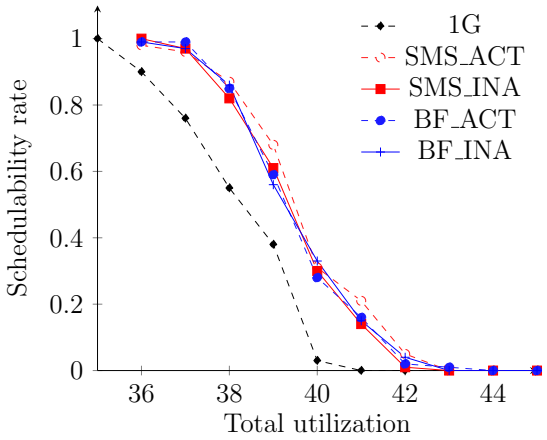


Figure 3.5: Schedulability rate for 200 tasks, with $prm = 50\%$.

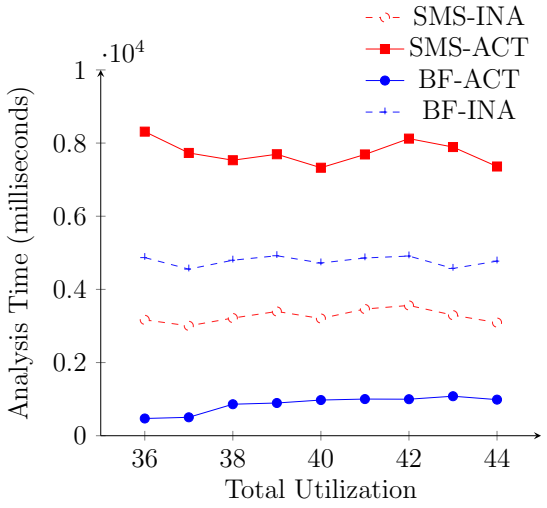


Figure 3.6: Analysis time as a function of total utilization.

3.7 Summary

With this work, our goal is to leverage existing research and NVIDIA CUDA developments that facilitate the *spatial partitioning* of GPUs, and propose partitioning heuristics for real-time workloads with strict timing constraints. We propose a *policy* to partition a GPU into a number of sets or *partitions*, and assign tasks to these sets to achieve low interference while ensuring that all tasks (kernels) meet their timing requirements. We model the execution time of typical GPU workloads based on their classification as memory or compute intensive kernels and evaluate our heuristics over 100 simulation runs for each obtained data point. Our results show significant improvement in schedulability over the baseline execution

wherein the GPU is not partitioned. As future work, we will propose more sophisticated models to quantify inter-kernel interference to guide our heuristics. We also plan to port our solution onto existing platforms like the NVIDIA A100, which is currently the only NVIDIA platform that supports spatial partitioning. As GPUs become more powerful and prevalent, it is crucial to explore new methods to enable predictable multi-tasking on the GPU. The partitioning mechanism we propose here is inspired by CPU-based partitioning solutions, [47]. In Chapter 4, we explore a mechanism for multi-tasking in GPUs that exploits a feature unique to GPUs, namely, *fine-grained multi-threading*.

CHAPTER 4

EXPLORING FINE-GRAINED MULTI-TASKING

4.1 Introduction

Over the years, there has been significant research in the real-time community to develop mechanisms of providing real-time guarantees to GPU-using tasks through spatial partitioning (SP) [24]. In this work, we explore a fine-grained approach to enforce task priorities in the GPU. With SP, real-time performance comes at the cost of GPU utilization and energy consumption since per-SM power/clock gating is not available in current NVIDIA GPUs [48, 49]. Thus, instead of allocating disjoint partitions to different kernels, we propose arbitration among the kernels at a finer granularity, namely, at the *warp scheduler* level.

Both CPU and GPU architectures aim to improve their performance (cycles per program) by *latency-hiding*. This means that anytime an instruction stalls, the processor executes another instruction instead of *waiting*. This technique reduces the number of cycles taken to execute a given program. CPU microarchitectures hide latencies through techniques like out-of-order execution and/or fetching more than 1 instruction per cycle (super-scalar or very long instruction word architectures), et cetera. GPUs, on the other hand, improve their performance through *low-overhead* context switching between *warps*. A warp is a set of 32 threads which execute the same instruction in lock-step (SIMT). Thus, anytime a warp stalls at an instruction, the *warp scheduler* chooses another warp to be executed. This phenomenon of low-overhead context switching to hide latencies is called *fine-grained multi-threading*. To achieve this, each warp scheduler maintains a *pool* of warps it can choose from every cycle. The warp scheduling policy determines which warp is issued every cycle.

The warp scheduling policy in NVIDIA GPUs is agnostic to the kernels that the warps belong to, which can be problematic in real-time systems where different kernels have differ-

ent priorities based on the scheduling algorithm. Therefore, in this work we propose a warp scheduling algorithm that takes the kernel priority into account.

We outline our contributions in Subsection 4.1.1. In Section 4.2, we provide a microarchitectural overview of the compute hierarchy of NVIDIA GPUs reverse-engineered by the research community. This gives us a clear understanding about what *exactly* happens in the GPU streaming multiprocessors (SMs) from the time a kernel is launched until it is finished. Next, we present warp scheduling algorithms proposed by other research works in Section 4.3. We discuss our proposed algorithm and implementation on GPGPU-Sim in Section 4.4, followed by our results in Section 4.5.

4.1.1 Summary of Contributions

Our contributions are as follows:

- We run experiments on recent NVIDIA GPUs to derive conclusions on the warp scheduling algorithm used in existing NVIDIA GPUs.
- We propose a budget-based warp scheduling policy that uses information about the kernel priority.
- We implement and evaluate our proposed warp scheduling policy on GPGPU-Sim against state-of-the-art warp scheduling algorithms deployed on NVIDIA GPUs, namely greedy-then-oldest and loose-round-robin.

4.2 Background: Compute Hierarchy of GPUs

In this section, we summarize relevant details of the GPU compute hierarchy that are not shared by NVIDIA but have been reverse-engineered by the research community. We believe it is important to know the *microarchitecture* details of the GPU compute hierarchy before delving into the finer details involving warp scheduling.

Any given kernel with an arbitrary kernel configuration goes through multiple hierarchies of arbitration in the GPU [22]. First, the thread *blocks* of the kernel are distributed to the

available SMs. For the blocks dispatched onto an SM, the *warps* of each block are distributed among the warp schedulers of the SM. Finally, the warp scheduler arbitrates between the warps assigned to it and determines which warp will issue an instruction every cycle. Details of each level of arbitration are described as follows.

Thread-Block to SM Mapping In NVIDIA GPUs, SM identifiers (*smid*s) range from 0 to the total number of SMs minus one. When a kernel is launched, the blocks are distributed to the SMs in a round-robin manner, starting with all the even indexed SMs and then proceeding to the odd-indexed SMs. Before mapping a thread block to an SM, a check is done if the SM has enough resources to execute the SM. If no SMs are free, the block waits until an SM has enough resources to run it. There are some caveats to this policy, covered in [22]. In the case of concurrent kernel execution, when multiple kernels are launched in different *CUDA streams*, the kernel launched first occupies all the SMs required by its thread blocks. Therefore, when the second kernel is launched, it can only occupy the *left-over* SM resources, and this is called the *left-over policy* [50, 22]. The grid of the second kernel is launched only if there are enough resources *left* to allow the execution of at least one thread block of the second kernel. As a consequence, many concurrently launched kernels end up being executed sequentially, depending on the GPU resources [50]. Thus, in order to ensure that the thread blocks of a higher priority kernel are *actually* scheduled on an SM, a budget on the maximum number of warps per kernel in each SM needs to be enforced. In our work, we configure the test kernels to occupy *half* of each SM. We will implement the budget-based warps per kernel limit in each SM as a part of future work.

Inside an SM: Warp to Warp Scheduler Mapping An SM can consist of multiple *warp schedulers*. Figure 2.2b shows an arbitrary number of schedulers per SM. This number depends on the specific GPU architecture; for example, the Volta and Turing microarchitectures have four schedulers per SM [51]. The scheduler identifiers (*sched_id*) within an SM range from 0 to the number of schedulers (*num_sched*) minus one. A warp with a given warp identifier (*warp_id*) is mapped to a scheduler based on, $sched\_id = warp\_id \ \% \ num\_sched$ [16, 51].

Inside an SM: Within a Warp Scheduler Each scheduler has a *fetch unit* that fetches one or more instructions per cycle from the instruction cache. Decoded instructions are stored in the instruction buffer or *I-Buffer*. The instruction buffer is statically partitioned to store a fixed number of instructions per warp [16]. Every cycle, the warp scheduler chooses which warp to issue from the pool of *ready warps*. Ready warps are warps whose next instruction is ready to be issued, i.e. they are not waiting on any previous instructions' results. The warp scheduler issues the instruction stored in the I-Buffer entry associated with the *selected* warp and dispatches it down the pipeline to the corresponding ALU depending on the instruction type. Research shows that the NVIDIA Fetch Unit schedules instruction cache access between the warps in a *round-robin* fashion [16, 52, 53]. Based on the related work, the policy used by the Warp Scheduler remains unclear. Some work [22, 54] claims that the policy used by the scheduler is *loose-round-robin (LRR)*. In the LRR policy, the warps are issued in a round-robin policy and if a warp is not *ready* during its turn, then the next warp in the round robin order is issued. Other work [16, 55] claims that the NVIDIA warp scheduler issues instructions using a *greedy-then-oldest (GTO)* policy. In the GTO policy, the same warp is issued by the warp scheduler every cycle until it eventually stalls, at which point the warp scheduler chooses the *oldest ready* warp. It is easy to see that LRR assigns *equal* priority to all warps, ensuring that all warps make equal progress. LRR is beneficial if the warps in the SM have spatial locality and share cache lines and DRAM row buffers, thus increasing cache and row buffer hits. However, the downside is that if there is no inter-warp locality, all the warps will reach long latency operations at the same time and there will be no warps left to hide this latency, resulting in idle cycles and performance loss. Additionally, in the case of real-time systems, warps belonging to different kernels (with different *real-time* priorities) will be given the *same* priority, which could lead to a higher-priority kernel missing its deadline, at the cost of *fairness*. GTO attempts to overcome the problem due to long latency instructions by letting warps to make unequal progress. With GTO, the long-latency periods of the warps do not overlap, ensuring that there are always enough warps to hide a long latency stall. From a real-time perspective however, GTO can be more catastrophic than LRR, since GTO gives a higher priority to *older* warps. Thus, if a lower-priority kernel is launched before a higher priority one, GTO will always

prioritize the warps of the lower-priority kernel, significantly increasing the response time of the higher priority kernel, ultimately leading to a deadline miss. We propose a warp scheduling policy that gains from the benefits of GTO but also takes the kernel real-time priorities into account, giving us the best of both worlds – performance and predictability.

4.3 Related Work

The goal of a warp scheduling policy is to effectively hide high latencies, usually caused by memory instructions. Whenever one warp stalls, the scheduler picks another warp that is ready issue its next instruction. Therefore, the second warp *hides* the latency of the first warp. However, there may be cycles where all warps are waiting on long latency operations, leaving no warp ready to be issued, resulting in an idle cycle and thereby worsening performance. Many warp scheduling policies have been proposed by the research community to solve this problem. The two widespread policies adopted in GPUs are GTO and LRR. Two-level warp scheduling is proposed in [56, 57]. The two-level warp scheduler maintains the warps as two subgroups to improve performance [56] and energy efficiency [57]. One subgroup is referred to as a fetch group and another subgroup called the ready queue. The scheduler only selects warps from the ready queue for execution. A warp in the ready queue is demoted to the fetch group when it encounters a long latency instruction. LRR and GTO policies can be used to order warps between and within the groups. Jog et al. [58] improved upon the two-level warp scheduler by assigning warps with continuous identifiers to different groups, to reduce L2 and DRAM bank conflicts between adjacent warps. While it is evident that different scheduling policies are suitable for different workloads, [59] leverages this information to determine at compile time, which warp scheduling policy to apply on the different phases of the kernel. The work in [60] extends this to be dynamic, i.e. based on the instruction issue pattern at runtime. Other work [61] proposes modulating the warp scheduling policy to shape the cache access patterns to avoid cache thrashing, and subsequent misses altogether.

All of the above mentioned solutions aim to improve the instructions per cycle and utilization of the GPU by preventing idle SM cycles. However, all of these policies are kernel

agnostic. While all these policies perform reasonably well in the case of homogeneous warps (warps belonging to a single kernel), minimal investigation has been done on what happens in the case of heterogeneous warps, and how the performance of each kernel is impacted. Additionally, in the case of real-time systems where real-time priorities are associated with each kernel, it would be disastrous if the kernel with the higher real-time priority is given lower priority by the warp scheduler. In this work, we examine the performance of GTO and LRR when warps of two different kernels are arbitrated by the warp scheduler. Furthermore, we believe we are the first to propose a warp scheduling algorithm that is aware of the real-time priorities of the kernels associated with the warps, and use this knowledge to achieve predictable execution when multi-tasking on GPUs.

4.4 Proposed Warp Scheduling Algorithm

We implement the warp scheduling algorithm on GPGPU-Sim. Therefore, we first elaborate on the existing warp scheduling framework in GPGPU-Sim. This is followed by the description of our proposed scheduling policy and finally an example to illustrate how it works.

4.4.1 Warp Scheduling Framework in GPGPU-Sim

The issue unit in GPGPU-Sim attempts to issue an instruction into the functional units of the SM every cycle. The tasks performed by the issue unit are outlined in Algorithm 4. The issue unit calls the **IssueWarp** procedure every simulation cycle. Within the procedure, it first calls the **SortWarps** function to order the warps based on the warp scheduling policy. The existing simulation framework has different implementations (LRR, GTO, et cetera) for this function depending on the configuration of the simulator. Once the issue unit has a prioritized queue of warps, called Q_{issue}, it iterates through the warps until it finds a *ready* warp and the functional units (special-function unit, load-store Unit, tensor core, et cetera) required by the instruction of the warp is available. The issue unit dispatches the warp to the respective functional unit and returns. It is again invoked in the next cycle.

Algorithm 4 Issue Unit

```
1  state W_greedy                                   ▷ Warp issued in last cycle
2  state Q_prev                                     ▷ Prioritized queue of warps in the last cycle
3  state Warps                                      ▷ List of warps assigned to this scheduler
4  procedure IssueWarp
5      Q_issue ← SortWarps(W_greedy, Q_prev, Warps)
6      nop ← True                                   ▷ NOP if no warp issued
7      for warp ∈ Q_issue do
8          if warp is ready and functional unit is available then
9              Dispatch(warp)                       ▷ Dispatches warp to functional unit
10             W_greedy ← warp
11             nop ← False
12             break
13         end if
14         Q_prev ← Q_issue
15     end for
16 end procedure
```

4.4.2 Real-Time Priority Aware Warp Scheduling (RT-PAWS)

We now discuss the implementation of RT-PAWS, that is, the **SortWarps** function called by the issue unit each cycle. In this policy, we assume that there can be warps from a maximum of two different kernels in every scheduler. Thus, if we combine all warps with the same priority into a *group*, a maximum of 2 groups will be created. If only a single kernel is launched (at start-up, for example), we fall back to the GTO Policy. Let us consider the case wherein there are two groups, each containing warps of different kernels with different *real-time priorities*. We start with the following definitions used in the algorithm:

- Q_{issue}, the prioritized queue of warps generated by the scheduling algorithm for the Issue Unit in *this* cycle.

- Q_{prev}, the prioritized queue of warps that the scheduling algorithm had generated in the *previous* cycle.

- W_{greedy}, the warp that is issued in the last cycle. Note that this may or may not be the head of Q_{prev}. If the head of Q_{prev} (called W_{headp}) stalled for any reason, the Issue Unit would have attempted to issue the next warp in the queue and so on, until it found a warp that *can* be issued (Algorithm 4). Thus, if W_{greedy} matches the head of

Q_{prev}, we can conclude that the head did not stall. This is done in Lines 14 and 15 of algorithm.

- g_0 and g_1, the two groups of warps corresponding to the different warp priorities. The algorithm always generates the queue of prioritized warps, and we read Q_{prev} to be in the form $(g_0 \oplus g_1)$ where $\oplus$ is list concatenation. g_0 is the group whose warps were inserted first in Q_{prev}. The warps of g_1 were inserted after all warps of g_0. Thus, intuitively, one can expect that a warp from g_0 would have been issued by the Issue Unit in the previous cycle. The only case a warp from g_1 would be issued is if all warps in g_0 were stalled.

- *budget*, the budget associated with each warp group. A budget for each group is necessary to ensure that the group with the lower priority is not starved. This budget is initialized to the priority of the kernel that the warps belong to. Here we assume that the warp priority equals the kernel's real-time priority, but this can be changed based on the application. If the budget of a particular group is zero, it can never be prioritized by the algorithm, and warps belonging to that group would only be issued if all the warps of the other group stall.

Each cycle, there are two groups, g_0 and g_1. The scheduling algorithm has two functions: (i) update any group budgets if needed, and (ii) determine the relative order between the groups and the order within the groups. We consider the following cases to determine the behavior of the scheduler.

1. **The head of Q_{prev} did not stall.** (Line 15) The scheduler continues inserting warps of g_0 first in a GTO manner, followed by warps of g_1 sorted by a oldest warp first (OLD) policy (Line 16).

2. **The head of Q_{prev} stalled and a warp from g_0 was issued.** (Line 18) The budget of g_0 is decremented by 1 (Line 19). If the budget of g_0 is still non-zero, the scheduler continues inserting warps of g_0 first in a GTO manner, followed by warps of g_1 sorted by a oldest warp first (OLD) policy (Line 23). However, if the budget of g_0 becomes

Algorithm 5 Sort Warps with Real-Time Priority Aware Warp Scheduling (RT-PAWS)

```
1  state groups                          ▷ List of groups; warps with same priority in one group
2  input W_greedy                        ▷ Warp issued in last cycle
3  input Q_prev                          ▷ Prioritized queue of warps in the last cycle
4  input Warps                           ▷ List of warps assigned to this scheduler
5  output Q_issue                        ▷ Warps sorted with policy
6  function SortWarps(W_greedy, Q_prev, Warps)
7      if length(groups) < 2 then        ▷ If zero or one groups
8          return GTO(Warps, W_greedy)   ▷ Resort to GTO
9      end if
10     assert length(groups) = 2         ▷ Assume maximum of two groups
11     (g_0, g_1) ← groups
12     W_headp ← head of Q_prev
13     head_priority ← GetPriority(g_0)
14     head_stalled ← W_headp ≠ W_greedy ▷ head_stalled is Boolean
15     if not head_stalled then
16         return GTO(g_0, W_greedy) ⊕ OLD(g_1)   ▷ ⊕: list concatenation
17     end if
18     if head_priority = GetPriority(W_greedy) then
19         DecrementBudget(g_0)          ▷ Reduce budget of warps ∈ g_0 by 1
20     end if
21     head_budget ← GetBudget(g_0)
22     if head_budget > 0 then
23         return GTO(g_0, W_greedy) ⊕ OLD(g_1)
24     else
25         ResetBudget(g_0)
26         return OLD(g_1) ⊕ OLD(g_0)    ▷ Swap group orders
27     end if
28 end function
29
30 function ResetBudget(WarpList)
31     for warp ∈ WarpList do
32         warp.budget ← warp.priority
33     end for
34     return
35 end function
36
37 function GTO(WarpList, W_greedy)
38     return warp list sorted by Greedy-Then-Oldest
39 end function
40
41 function OLD(WarpList)
42     return warp list sorted by Oldest Warp ID
43 end function
```

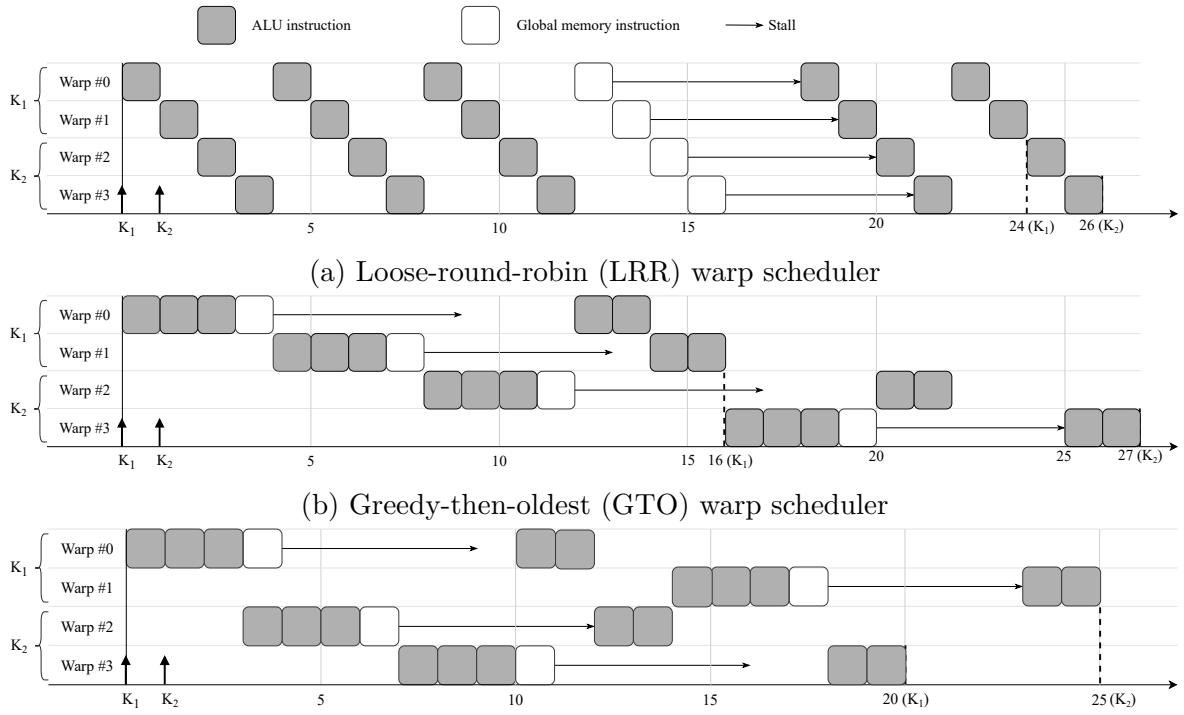
zero, the scheduler inserts the warps of g_1 first using the OLD policy, followed by warps of g_0 sorted by the OLD policy (Line 26).

3. **The head of Q_{prev} stalled and a warp from g_1 was issued.** If the budget of g_0 is non-zero (Line 22), the scheduler continues inserting warps of g_0 first in a GTO manner, followed by warps of g_1 sorted by a oldest warp first (OLD) policy. If the budget of g_0 becomes zero (Line 24), the scheduler inserts the warps of g_1 first using the OLD policy, followed by warps of g_0 sorted by the OLD policy (Line 26).

4.4.3 Example

Consider two kernels K_1 and K_2, where K_2 has a higher real-time priority than K_1 but is launched after K_1. We assume an SM warp scheduler with just 4 warps where warps 0 and 1 belong to K_1 and warps 2 and 3 belong to K_2. We consider a short example kernel of only 6 instructions.

Figure 4.1 illustrates the execution order for LRR, GTO, and RT-PAWS. GTO introduces *priority inversion* since it executes the warps of lower priority kernel K_1 first (since warps of K_1 are older). Thus, K_2 finishes last with GTO, which is disastrous since it is the highest priority kernel. In Figure 4.1a, with the LRR policy, both kernels suffer since all warps encounter the long latency instruction at the same time. We see that K_2 finishes earliest with RT-PAWS in Figure 4.1c. Note that K_2 would have finished earlier if the warp priority of K_1 was set to 0 instead of 1. The average execution time of K_1 and K_2 is the highest in LRR (25 cycles), whereas the average execution time with RT-PAWS (22.5 cycles) is one cycle more than GTO (21.5 cycles), which is not significant but worth noting. However, the finish time of both kernels, i.e. $max(K_1, K_2)$, is the least with RT-PAWS, indicating that RT-PAWS achieves the maximum system throughput. We witness very similar results for real-world benchmarks discussed in Section 4.5. Note that K_2 with RT-PAWS does not finish as early as K_1 with GTO; this is because warp 0 from K_1 is executed greedily in the first few cycles until it stalls.



(a) Loose-round-robin (LRR) warp scheduler

(b) Greedy-then-oldest (GTO) warp scheduler

(c) Real-time priority aware warp scheduler (RT-PAWS). Warp priority of $K_1 = 1$ and $K_2 = 2$.

Figure 4.1: An example illustrating the execution order to LRR, GTO and RT-PAWS. K_1 has a lower real-time priority than K_2. K_1 is launched first at $t = 0$ and K_2 at $t = 1$.

4.5 Evaluation

We evaluate the real-time and throughput performance of the proposed real-time priority based warp scheduling policy (RT-PAWS) against loose-round-robin (LRR) and greedy-then-oldest (GTO). We discuss the experimental setup on GPGPU-Sim in Subsection 4.5.1 and the results in Section 4.5.2. As discussed in Section 4.2, a consensus about the warp scheduling policy used on NVIDIA hardware is still not reached in the research community. Many works [16, 52, 53] claim that it is LRR and many claim [16, 55] that it is GTO, therefore we compare performance against both.

4.5.1 Methodology

We evaluate the proposed RT-PAWS policy on GPGPU-Sim. The details of the simulator configuration are described in Table 4.1. Note that we used a configuration to model the NVIDIA TITAN V GPU, as it was the most recent tested configuration released by GPGPU-

Table 4.1: Details of the configuration used in GPGPU-Sim.

Description	Configuration
Device	NVIDIA TITAN V
GPU Clusters	40
SMs per Cluster	2
Total SMs	80
Schedulers per SM	4
Warp Scheduler	Either GTO, LRR, or RT-PAWS
L1 + Shared Memory Size	128KB
L2 Size	4.5MB

Sim. We evaluate the proposed policy on all benchmarks in Table 4.2. **pc** is a purely compute benchmark wherein each thread executes only one load and one store instruction, and loops over compute (non-memory) instructions a million times. For each benchmark in Table 4.2, we launch two instances of the same benchmark and configure the GPU and kernel configuration to ensure that every warp scheduler in the GPU has an equal number of warps from the first kernel instance (subsequently referred to as K_1) and the second kernel instance (subsequently referred to as K_2). K_1 and K_2 are launched on different streams to ensure concurrent execution. We verify this through the simulation logs as well. We believe that launching two instances of the same kernel (as opposed to heterogeneous kernels) constructs the *worst-case scenario* since both kernels K_1 and K_2 compete for the same resources. In the kernel code, K_1 is launched **before** K_2. We assume here that K_2 has a higher real-time priority than K_1, since that is the more pessimistic case. With this setup we evaluate (i) the response times for both K_1 and K_2 when scheduled with LRR, GTO and RT-PAWS, compared. Furthermore, we also compare the average execution time of K_1 and K_2 to demonstrate the effect of RT-PAWS on the system throughput. These results and details of how we picked the warp priorities for K_1 and K_2 when scheduled with RT-PAWS are discussed in Section 4.5.2.

4.5.2 Results

Response Times We compare the response times of kernel invocations K_1 and K_2. To ensure that all benchmarks' response times are in the scale, we normalize the response times

Table 4.2: Benchmarks used in Figure 3.1 and Figure 3.2. MI indicates memory intensive and CI indicates compute intensive.

Benchmark	Description	Type
pc	Purely Compute	CI
pf	Pathfinder [40]	CI
2dconv	2D Convolution [62]	CI
dxtc	DXTC [39]	CI
vadd	Vector Add [39]	MI
gemm	Matrix Multiply [62]	MI
hist	Histogram [39]	MI
atax	$A^T AX$ [62]	MI

by the K_1 response time in LRR. We choose LRR since the current NVIDIA GPUs most probably implement LRR warp schedulers [16, 52, 53]. When implementing the proposed RT-PAWS policy, recall that we need to set the warp priorities for K_1 and K_2. We assume that the kernel task K_2 is higher priority. Therefore, we pick the priority of K_1 warps as 1, and the priority of K_2 warps is assigned as p where $p \in \{2, 4, 8\}$. While all these three values give similar performance, we choose the p that gives the lowest response time for K_2. The *exact* relationship between the warp priority and kernel execution time needs to be investigated further, and will be studied in future work.

Figure 4.2 shows the response time results. We see that RT-PAWS *always* outperforms LRR for both K_1 and K_2. The response time of K_1 is lower with the GTO warp scheduler for histogram and atax (memory intensive kernels). This is because K_1 is launched *first*. Hence, all the K_1 warps are *older* than K_2 warps and are hence implicitly prioritized by the scheduling policy. This is a case of *priority inversion* since the warps of the kernel which has a lower *real-time* priority (K_1) are being prioritized over the warps of the kernel with the higher *real-time* priority. RT-PAWS tries to bridge this anomaly by letting the developer explicitly assign the warp priorities according to the *real-time* priority of the kernel. Therefore, the K_2 response times are lower when scheduled using RT-PAWS. However, the K_2 response times with RT-PAWS are not as low as the K_1 response times in GTO because the K_1 warps still enter the SM first and are executed greedily (until they stall) even when K_2 warps have entered the SM. We refer the reader to the example in Subsection 4.4.3 which clearly illustrates this phenomenon.

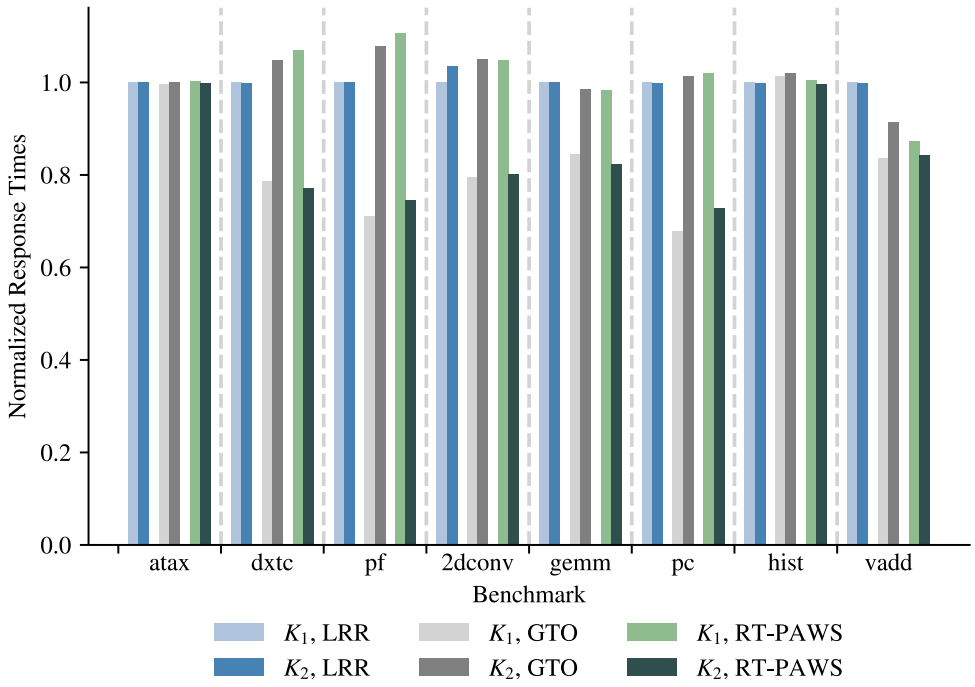


Figure 4.2: Response times for kernel launches K_2 and K_1, for different benchmarks, under various warp scheduling policies. The response times for every benchmark are normalized with respect to the response time of K_1 under the LRR policy.

System Throughput Figure 4.3 shows the average execution time of K_1 and K_2 normalized with respect to LRR. A lower average execution time indicates a higher system throughput. RT-PAWS outperforms LRR. RT-PAWS is as good as GTO for most kernels. This is because RT-PAWS uses GTO to order warps within a warp group. Therefore, with RT-PAWS, we get the benefits of GTO (high throughput) but none of the issues (priority inversion).

4.6 Summary

In this work, we explore another mechanism to facilitate predictable GPU multi-tasking. We propose enforcing a priority-based scheduling policy at the finest granularity of GPU execution, i.e. at the warp level. Our priority-based warp scheduler is evaluated on the state-of-the-art NVIDIA GPU Simulator GPGPU-Sim. Results show that the response times of higher priority tasks reduce significantly, even when they are launched after a lower

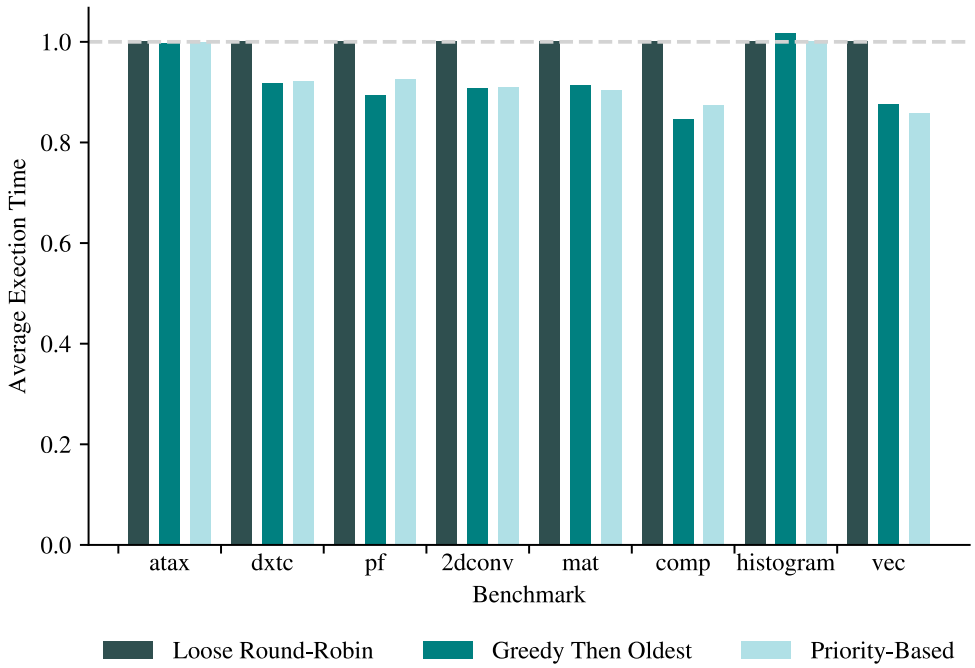


Figure 4.3: Average execution time of K_1 and K_2 for each benchmark, normalized with respect to the average execution time with LRR. Lower is better.

priority task. This is in contrast with other state-of-the-art warp scheduling policies like loose-round-robin (used in current NVIDIA GPUs) and greedy-then-oldest which suffer from significant priority inversion due to high blocking times of lower priority kernels launched earlier. Moreover, the throughput of our proposed policy is higher than LRR and as good as (on average) the throughput achieved by GTO. Given the positive results of our evaluations, we believe that this is a viable solution to achieve higher utilization and better schedulability for hard and soft real-time systems. As future work, we will extend our warp scheduling algorithm to include more than two warp priorities, in order to scale with any number of concurrently executed kernels. Understanding the relationship between warp priorities and the overall performance of the algorithm will also be thoroughly studied and modeled. Our solution proposes a priority-based issue unit in the SMs; however, the fetch order of warps (i.e. fetching instructions from the instruction cache) is still round-robin in all NVIDIA GPUs. Therefore, we will also investigate scheduling policies for the fetch unit inside GPU SMs.

CHAPTER 5

CONCLUSION

In this thesis, we investigate two orthogonal approaches to enable multi-tasking on GPUs with the primary goal of predictable execution. The first approach is spatial partitioning (SP) where tasks are allocated to disjoint compute partitions of the GPU. There has been considerable progress in the research community as well as industry on methods to implement SP on modern GPUs, but harnessing the potential of SP to improve utilization and predictability has been scarcely investigated. Therefore, we attempt to determine in what way a GPU must be partitioned, to achieve high utilization while also respecting real-time constraints. We propose partitioning heuristics that, given a task set, generate a number of partitions and allocate tasks to these partitions, optimizing for improved utilization with the constraint that all tasks meet their deadlines.

The second approach to enable multi-tasking is simultaneous multi-kernel (SMK), where arbitration between tasks is not done at the partition or compute core granularity, but is in fact done at the lowest level of GPU execution, specifically, at the warp-level. We propose a priority-based warp scheduling policy that aims to improve the response time of real-time tasks, and eliminates priority inversion. We show that our policy not only improves the response time but also has comparable or better throughput performance than the state-of-the-art warp scheduling policies used in modern NVIDIA GPUs. With our proposed warp scheduling policy, coupled with the ability to specify priorities associated with warps, or kernels, we believe that a variety of application domains can harness the computing power of GPUs by setting priorities appropriately. Therefore, it is our hope that this work motivates more research in this direction and eventually encourages manufactures like NVIDIA to include a mechanism to specify "warp priorities" in their software API. Such a mechanism would create a hardware solution that can effectively be utilized by throughput-oriented

systems as well as real-time systems through the means of a more powerful and expressive API.

While both of our solutions are proposed in isolation, our aim is to integrate them. In fact, a good solution would be a combination of SMK and SP. Intuitively, kernels with disjoint resource requirements could be scheduled in the same partition and the priority-based warp scheduling policy would arbitrate between them. Our goal moving forward is to propose a hybrid solution combining SP and SMK to achieve maximum throughput and predictability in GPUs.

REFERENCES

[1] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," in *2011 IEEE 32nd Real-Time Systems Symposium*, 2011, pp. 217–226.

[2] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni, "PALLOC: Dram bank-aware memory allocator for performance isolation on multicore platforms," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014, pp. 155–166.

[3] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013, pp. 55–64.

[4] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for cots-based embedded systems," in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2011, pp. 269–279.

[5] A. Bansal, J. Singh, Y. Hao, J.-Y. Wen, R. Mancuso, and M. Caccamo, "Reconciling predictability and coherent caching," in *2020 9th Mediterranean Conference on Embedded Computing (MECO)*, 2020, pp. 1–6.

[6] M. Alcon, H. Tabani, L. Kosmidis, E. Mezzetti, J. Abella, and F. J. Cazorla, "Timing of autonomous driving software: Problem analysis and prospects for future solutions," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020, pp. 267–280.

[7] M. Huzaifa, R. Desai, S. Grayson, X. Jiang, Y. Jing, J. Lee, F. Lu, Y. Pang, J. Ravichandran, F. Sinclair, B. Tian, H. Yuan, J. Zhang, and S. V. Adve, "Exploring extended reality with ILLIXR: A new playground for architecture research," arXiv preprint arXiv:2004.04643, 2021.

[8] C. Xie, X. Zhang, A. Li, X. Fu, and S. Song, "PIM-VR: Erasing motion anomalies in highly-interactive virtual reality world with customized memory cube," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 609–622.

[9] P. E. McKenney, "'Real time' vs. 'real fast': How to choose?" presented at Ottawa Linux Symposium, July 2008.

[10] R. Smith, "NVIDIA Ampere unleashed: NVIDIA announces new GPU architecture, A100 GPU, and accelerator," 2020. [Online]. Available: https://www.anandtech.com/show/15801/nvidia-announces-ampere-architecture-and-a100-products

[11] H. Kim, P. Patel, S. Wang, and R. R. Rajkumar, "A server-based approach for predictable GPU access with improved analysis," *Journal of Systems Architecture*, vol. 88, pp. 97–109, 2018.

[12] G. A. Elliott, B. C. Ward, and J. H. Anderson, "GPUSync: A framework for real-time GPU management," in *2013 IEEE 34th Real-Time Systems Symposium*. IEEE, 2013, pp. 33–44.

[13] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, "Simultaneous multikernel GPU: Multi-tasking throughput processors via fine-grained sharing," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 358–369.

[14] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar, "RGEM: A responsive GPGPU execution model for runtime engines," in *2011 IEEE 32nd Real-Time Systems Symposium*. IEEE, 2011, pp. 57–66.

[15] J. Zhong and B. He, "Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1522–1532, 2014.

[16] M. Khairy, A. Jain, T. M. Aamodt, and T. G. Rogers, "A detailed model for contemporary GPU memory systems," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019, pp. 141–142.

[17] NVIDIA Drive AGX product website, 2021. [Online]. Available: https://www.nvidia.com/en-us/self-driving-cars/drive-platform/hardware/

[18] H. Mujtaba, "NVIDIA GeForce gained discrete GPU market share in Q4 2020 versus AMD Radeon," 2021. [Online]. Available: https://wccftech.com/nvidia-geforce-gained-discrete-gpu-market-share-in-q4-2020-versus-amd-radeon/

[19] N. Otterness and J. Anderson, "AMD GPUs as an alternative to NVIDIA for supporting real-time workloads," in *ECRTS*, 2020.

[20] M. Yang, N. Otterness, T. Amert, J. Bakita, J. Anderson, and F. D. Smith, "Avoiding pitfalls when using NVIDIA GPUs for real-time tasks in autonomous systems," in *ECRTS*, 2018.

[21] A. Klöckner, "PyCUDA: Even simpler GPU programming with Python," presented at Courant Institute of Mathematical Sciences, New York University, Sept. 22, 2010.

[22] I. S. Olmedo, N. Capodieci, J. L. Martinez, A. Marongiu, and M. Bertogna, "Dissecting the CUDA scheduling hierarchy: A performance and predictability perspective," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020, pp. 213–225.

[23] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, 3rd ed. Springer Publishing Company, Incorporated, 2011.

[24] S. Jain, I. Baek, S. Wang, and R. Rajkumar, "Fractional GPUs: Software-based compute and memory bandwidth reservation for GPUs," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019, pp. 29–41.

[25] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni, "A survey on cache management mechanisms for real-time embedded systems," *ACM Comput. Surv.*, vol. 48, no. 2, Nov. 2015. [Online]. Available: https://doi.org/10.1145/2830555

[26] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte, "The case for GPGPU spatial multitasking," in *IEEE International Symposium on High-Performance Comp Architecture*, 2012, pp. 1–12.

[27] X. Zhao, Z. Wang, and L. Eeckhout, "Classification-driven search for effective SM partitioning in multitasking GPUs," in *Proceedings of the 2018 International Conference on Supercomputing*, ser. ICS '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3205289.3205311 p. 65–75.

[28] X. Zhao, M. Jahre, and L. Eeckhout, "HSM: A hybrid slowdown model for multitasking GPUs," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3373376.3378457 p. 1371–1385.

[29] W. Zhao, Q. Chen, H. Lin, J. Zhang, J. Leng, C. Li, W. Zheng, L. Li, and M. Guo, "Themis: Predicting and reining in application-level slowdown on spatial multitasking GPUs," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Brazil: IEEE, 2019, pp. 653–663.

[30] Q. Hu, J. Shu, J. Fan, and Y. Lu, "Run-time performance estimation and fairness-oriented scheduling policy for concurrent GPGPU applications," in *2016 45th International Conference on Parallel Processing (ICPP)*, 2016, pp. 57–66.

[31] T. Allen, X. Feng, and R. Ge, "Slate: Enabling workload-aware efficient multiprocessing for modern GPGPUs," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 252–261.

[32] Q. Sun, Y. Liu, H. Yang, Z. Luan, and D. Qian, "SMQoS: Improving utilization and energy efficiency with QoS awareness on GPUs," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, 2019, pp. 1–5.

[33] S. K. Saha, Y. Xiang, and H. Kim, "STGM: Spatio-temporal GPU management for real-time tasks," in *25th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2019, Hangzhou, China, August 18-21, 2019*. IEEE, 2019. [Online]. Available: https://doi.org/10.1109/RTCSA.2019.8864564 pp. 1–6.

[34] S. Pai, M. Jacob, and R. Govindarajan, "Improving GPGPU concurrency with elastic kernels," vol. 48, 03 2013, pp. 407–418.

[35] H. Dai, Z. Lin, C. Li, C. Zhao, F. Wang, N. Zheng, and H. Zhou, "Accelerate GPU concurrent kernel execution by mitigating memory pipeline stalls," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 208–220.

[36] S.-K. Shekofteh, H. Noori, M. Naghibzadeh, H. S. Yazdi, and H. Fröning, "Metric selection for GPU kernel classification," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 4, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3295690

[37] Y. Ukidave, F. N. Paravecino, L. Yu, C. Kalra, A. Momeni, Z. Chen, N. Materise, B. Daley, P. Mistry, and D. Kaeli, "NUPAR: A benchmark suite for modern GPU architectures," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2668930.2688046 p. 253–264.

[38] V. Volkov, "Understanding latency hiding on GPUs," PhD dissertation, University of California Berkeley, 2016.

[39] "CUDA samples." [Online]. Available: https://docs.nvidia.com/cuda/cuda-samples/index.html

[40] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2009, pp. 44–54.

[41] X. Mei and X. Chu, "Dissecting GPU memory hierarchy through microbenchmarking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 72–86, 2016.

[42] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[43] H. Zahaf, G. Lipari, M. Bertogna, and P. Boulet, "The parallel multi-mode digraph task model for energy-aware real-time heterogeneous multi-core systems," *IEEE Transactions on Computers*, vol. 68, no. 10, pp. 1511–1524, 2019.

[44] H.-E. Zahaf, A. E. H. Benyamina, R. Olejnik, and G. Lipari, "Energy-efficient scheduling for moldable real-time tasks on heterogeneous computing platforms," *J. Syst. Archit.*, vol. 74, p. 46–60, Mar. 2017.

[45] P. Emberson, R. Stafford, and R. I. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *Proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, 2010, pp. 6–11.

[46] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, 2008.

[47] R. Mancuso, R. Pellizzoni, M. Caccamo, L. Sha, and H. Yun, "WCET(m) estimation in multi-core systems using single core equivalence," in *2015 27th Euromicro Conference on Real-Time Systems*, 2015, pp. 174–183.

[48] M. Abdel-Majeed, D. Wong, and M. Annavaram, "Warped gates: Gating aware scheduling and power gating for GPGPUs," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 111–122.

[49] "Jetson AGX Xavier thermal design guide." [Online]. Available: https://static5.arrow.com/pdfs/2018/12/12/12/22/1/565659/nvda_/manual/jetson_agx_xavier_thermal_design_guide_v1.0.pdf

[50] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving GPGPU concurrency with elastic kernels," *SIGARCH Comput. Archit. News*, vol. 41, no. 1, p. 407–418, Mar. 2013. [Online]. Available: https://doi.org/10.1145/2490301.2451160

[51] Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza, "Dissecting the Nvidia Turing T4 GPU via microbenchmarking," arXiv preprint arXiv:1804.06826, 2019.

[52] N. B. Lakshminarayana and H. Kim, "Effect of instruction fetch and memory scheduling on GPU performance," in *Workshop on Language, Compiler, and Architecture Support for GPGPU*, vol. 88. Citeseer, 2010.

[53] Y. Liu, Z. Yu, L. Eeckhout, V. J. Reddi, Y. Luo, X. Wang, Z. Wang, and C. Xu, "Barrier-aware warp scheduling for throughput processors," in *Proceedings of the 2016 International Conference on Supercomputing*, 2016, pp. 1–12.

[54] S.-Y. Lee, A. Arunkumar, and C.-J. Wu, "CAWA: Coordinated warp scheduling and cache prioritization for critical warp acceleration of GPGPU workloads," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3S, pp. 515–527, 2015.

[55] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-Sim: An extensible simulation framework for validated GPU modeling," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 473–486.

[56] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving GPU performance via large warps and two-level warp scheduling," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: Association for Computing Machinery, 2011. [Online]. Available: https://doi.org/10.1145/2155620.2155656 p. 308–317.

[57] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-efficient mechanisms for managing thread context in throughput processors," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011, pp. 235–246.

[58] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "OWL: Cooperative thread array aware scheduling techniques for improving GPGPU performance," *SIGPLAN Not.*, vol. 48, no. 4, p. 395–406, Mar. 2013. [Online]. Available: https://doi.org/10.1145/2499368.2451158

[59] M. Awatramani, X. Zhu, J. Zambreno, and D. Rover, "Phase aware warp scheduling: Mitigating effects of phase behavior in GPGPU applications," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, 2015, pp. 1–12.

[60] M. Lee, G. Kim, J. Kim, W. Seo, Y. Cho, and S. Ryu, "iPAWS: Instruction-issue pattern-based adaptive warp scheduling for GPGPUs," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 370–381.

[61] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 72–83.

[62] L.-N. Pouchet, "Polybench v2.0," 2015. [Online]. Available: http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/